The Dissertation Committee for John Francis Bridgman, III
certifies that this is the approved version of the following dissertation:

Reliable Distributed Information: Agreement and Structure

Committee:

Vijay Garg, Supervisor

Ari Arapostathis

Sanjay Shakkottai

Mohamed Gouda

Sujay Sanghavi

Reliable Distributed Information: Agreement and Structure

by

John Francis Bridgman, III

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2018

Reliable Distributed Information: Agreement and Structure

Publication No. _____

John Francis Bridgman, III, Ph.D. The
University of Texas at Austin, 2018

Supervisor: Vijay Garg

The world is inherently distributed and concurrent. The more complicated systems become, the more likely they are to fail or partially fail. This work presents several results with Byzantine Agreement and some results of using coding in solving distributed and concurrent problems. We explore adding weights to processes to model *a priori* knowledge of process reliability. Then, some results of what can be done when performing repeated agreement. A result between combinatorial geometry and approximate Byzantine agreement is also provided. Coding is often used in communication, but here we provide examples of the usage of coding to minimize broadcast information and to solve a concurrent problem. The first use of coding is to notice the redundant information in distributed protocols and how to use a code to reduce the amount of information needed to be transmitted. The second is a method of using coding to provide a buffer of memory in a concurrent system that can be updated such that readers see the update as atomic.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

The world is inherently distributed and concurrent. The more complicated systems become, the more likely they are to fail or partially fail. Walk into a room and there is likely more processors in the room than there are people. For example, cellphones are ubiquitous and have more processing power than super computers did decades ago. When a system is distributed over hundreds of thousands of devices, the probability for one of them failing becomes a certainty. This brings up the question of how to deal with processes or nodes failing. How can we model all possible failures for a system? The worse case failure model is to assume that there is an all knowing adversary that controls all failed processes. This is the model used for Byzantine failures.

There has been much study about Byzantine failures in distributed systems. The problem that coined the term Byzantine in distributed systems was the Byzantine Generals (BG) problem introduced by Lamport, Shostak and Pease [35, 42]. There is extensive literature on the Byzantine Generals problem and its variations [1, 11, 13, 15, 20, 24, 32]. In the BG problem, there is a number of generals that may communicate using messengers. The generals wish to agree on whether or not they should attack or retreat. A few of the

generals are traitors and wish to force some to attack and other to retreat. If all non-traitor generals attack, then they will succeed. But, if some retreat or attack, then they will be routed. Then, the questions of the problem are: Can this problem be solved? If it can be solved, how many traitors can be tolerated? What are the minimum resources (messages, computation, time) required to solve the problem? The answer to these questions are now well known. If f processes are faulty, then $n \geq 3f + 1$ processes total are required and $f + 1$ rounds of communication are required in the worst case. A popular variation of this problem, called Byzantine Agreement (BA), has not one leader, but everyone proposes a value and all non-faulty processes must come to an agreement on the output. So, every process starts with an input value and the goal of the algorithm is to agree upon a common value in the presence of f arbitrary faulty processes. These two problems are equivalent because Byzantine Agreement and Byzantine Generals can be solved in terms of each other. The original problems are restricted to a one bit value, but there are several formulations of the problem where the values can be arbitrary bit streams. Another popular variation is Approximate Byzantine Agreement (ABA). ABA approximately agrees on a scalar value. In this variation, the processes propose a scalar value and then agree approximately on an output value. Fischer, Lynch, Paterson and Paterson [21] prove that even with one fault, BA is impossible in asynchronous systems. But, it turns out that if you allow for error between the values, then it is still possible to agree within that error in asynchronous systems. In general, all variations have constraints

similar to:

- Non-faulty processes consume a bounded amount of resources.
- Output values of non-faulty processes must satisfy a validity condition based on the input values of non-faulty processes.
- Output values of non-faulty processes must satisfy an agreement condition.

These three conditions for Byzantine Agreement become: (1) terminate in a bounded number of rounds, (2) the output must be proposed by a non-faulty process, and (3) all non-faulty processes agree on the same value.

Another case of agreement to consider is agreeing upon the order in which events happens. Serialized events and synchronization are things imposed upon the world by us. When a non-concurrent or centralized object exists, the object is that way because the object was designed to be that way. But, we, humanity, find reasoning about concurrent and distributed issues difficult. So, synchronization is imposed upon problems to make them tractable. The simplest type of synchronization is complete serialization, where only one action may take place at a time. The first processors used this sort of synchronization internally. After advances, ways of performing more than one task at a time where discovered. This underlines the importance of understanding synchronization in distributed and concurrent systems. Synchronization methods can be used in anything between scheduling when to meet a friend for lunch to ensuring the atomicity of a transaction in a database.

What exactly is synchronization? This work takes synchronization to mean an enforced "happens before" relationship between two events [34]. That is, the "happens before" relationship between the events is guaranteed to happen, regardless of all external input to the system or scheduling. A simple way to ensure that a "happens before" relationship for two events is for all actors in a system to agree upon the completion of one event before processing the next event. There are several ways to accomplish this. One could be to have a conductor, that signals intervals, such as a clock. To give an example, this is the purpose of a conductor in a symphony, to ensure that all musicians are kept synchronized with each other. Another example is an oscillator driving a processor. In both of these cases, the interval signal is a global state. Most distributed and concurrent systems assume some global clock, that gives a count of intervals that can be used to mark the passage of time in the system. In concurrent contexts, this is often the processor's memory bus clock. A single memory location can only be accessed by one device at a time, and this is enforced in hardware through things like cache consistency algorithms. So, in concurrent systems, synchronization is done in terms of memory access "happens before" relationships. In distributed systems, they either use a highly accurate low drift clock at each actor such as an atomic clock or make use of timeouts with a low accuracy clock where the timeout is chosen to be much larger than the clock synchronization error.

Another way to think about synchronization is to agree upon the "happens before" relationship rather than *a priori* enforce the relationship. One

simple way to agree upon a "happens before" is to time stamp all events with a logical clock. This clock does not need to be related to real time, but can be a logical clock. [34] Having a mark of some sort that indicates the passage of time is required for any communication that is not guaranteed to succeed [21]. Once an agreement algorithm is obtained in a system, all other synchronization methods can be implemented in terms of repeated agreement. This agreement may be partial or spread out over the system's idea of time. But, in the end, agreement on the order of events that require a "happens before" relationship must be made or no system can produce meaningful results.

1.1 System Models

The distributed execution model used in this paper is the standard reliable synchronous message passing model. Processes can only communicate by passing messages. Processes are assumed to be fully connected. The underlying system is assumed to be *synchronous*; i.e., there is an upper bound on the message delay and on the duration of actions performed by processes. The model assumes that processes may fail; but, the underlying communication system is reliable and satisfies first-in first-out (FIFO) message ordering. Message passing is assumed to be such that a process knows the identity of who sent the message. It is assumed that there is no source of randomness that cannot be influenced by an adversary. As such, only deterministic algorithms are considered. If faulty processes are allowed, they are assumed to be arbitrary. This includes faults as if an omniscient adversary controlled all faulty

processes. This work assumes that no secrets can be kept from an adversary, as such authenticated protocols such as public private key authentication are not usable.

The concurrent execution model used in this paper is based on typical multicore workstation running a modern preemptive multi-tasking operating system(OS). The model assumes that the OS scheduler makes the system appear in the absence of outside information as if there are an infinite number of cores. Each threads executes independently and can be interrupted by the operating system scheduler at any time to maintain the illusion of infinite cores. The model has a global shared memory region that all threads can access. Writes of more than one location to memory are assumed to not be atomic and may be interleaved with other writes and reads. The memory is assumed to be synchronous on some clock, and for each clock tick, only one thread may read or write a single location during that tick. It is assumed that if a completed write event A happens before some other read event B of the same location and no other write is concurrent with the interval from event A to event B; then, the read B will read the value that A wrote. The model puts no limit to the size of the individual memory locations. The location may be one bit in size or something larger.

1.2 Contributions

1.2.1 Weighted Byzantine Agreement

The Byzantine Agreement (BA) [35, 42] is a fundamental problem in distributed computing with extensive literature [1, 11, 13, 15, 20, 24, 32]. In the usual set-up, there are N processes required to agree on a common value, given that at most f of them may show arbitrary or Byzantine behavior.

In real-life applications, there may be multiple classes of processes. For example, in a mobile computing scenario, mobile hosts may be more likely to fail compared to mobile stations. In another example, processes in the same data center may be more likely to fail together. One generic way of dealing with different classes of processes is adversarial structure [30]. Adversarial structure is to enumerate every set of processes that may fail together. This work considers a simpler approach of assigning a weight to each process that represents some *a priori* knowledge of the reliability of that process.

Chapter 2 defines a weighted version of the Byzantine Agreement Problem (WBA) [25] and provides lower bounds and algorithms for it. In WBA, each process P_i is assigned a weight $w[i]$, such that $0 \leq w[i] \leq 1$ and the sum of all weights is 1. The WBA problem requires a protocol to reach consensus when the total weight of the failed processes is at most ρ. The weighted version gives some surprising results for the BA problem. First, even if greater than $N/3$ processes are Byzantine, the system can still reach consensus so long as ρ is less than $1/3$. This result is quite useful in the system with a small set of trusted processes and a large set of less trusted processes.

The Weighted Byzantine Agreement (WBA) problem can be specified as follows. All processes propose a binary value with the goal of deciding on one common value. Given a weight assignment to all processes, and the assumption that the weight of the processes that fail during the execution is at most ρ, the WBA problem is to design a protocol that satisfies the following conditions:

- **Agreement**: Two correct processes cannot decide on different values.
- **Validity**: The value decided must be proposed by some correct process.
- **Termination**: All correct processes decide in finite number of steps.

In Chapter 2, the *anchor* of a system (denoted by α_ρ) is defined as the least number of processes whose total weight exceeds ρ. WBA can be solved with the number of rounds equal to the system's anchor is shown. The anchor for a system with $\rho = f/N$ is $f + 1$ at most and, in many cases, is much smaller than $f + 1$. Two algorithms for the WBA problem are given: the weighted-Queen algorithm and the weighted-King algorithm. These algorithms are generalizations of the algorithms proposed by Berman and Garay [5] and Berman, Garay and Perry [4]. The weighted-Queen algorithm takes α_ρ rounds, each with two phases, and can tolerate any combination of failures so long as $\rho < 1/4$. The weighted-King algorithm takes α_ρ rounds, each with three phases, and can tolerate any combination of failures so long as $\rho < 1/3$.

1.2.2 Repeated Byzantine Agreement With Feedback

Chapter 3 presents Accurate Byzantine Agreement(ABA) with Feedback, a joint work with Bharath Balasubramanian and Vijay Garg [26,27]. In the standard version of Byzantine Agreement [15,20,24,35,42], the value that is agreed upon may be either of the binary values so long as it is proposed by at least one non-faulty process. In some scenarios, it is better for the system to agree on a specific value among the two binary values. For example, suppose in a distributed control system, a coordinated action needs to be taken (such as opening or closing a valve) depending upon the observations made by possibly faulty distributed processes. Depending upon the outcome of the action, the environment can provide a feedback if the action taken was correct or not.

The definition of ABA is as follows. Consider n processes consisting of non-faulty and faulty processes. There are multiple binary decisions that these n processes are required to make. For each possible decision (iteration of the ABA problem), each of the non-faulty processes proposes either 0 or 1. An algorithm that solves the Accurate Byzantine Agreement with Feedback (ABA) problem, must guarantee the following properties:

- Agreement: For each iteration, all non-faulty processes decide on the same value.

- Termination: The algorithm terminates in a finite number of rounds.

- Accuracy: The non-faulty processes agree on a value that is deemed correct by environmental feedback.

1.2.3 Vectorized Byzantine Agreement

Sometimes exact agreement is unnecessary, too expensive or the requirements are too difficult to satisfy. This is the motivation for Approximate Byzantine Agreement(ABA). For ABA the constraints are: (1) terminate in bounded time, (2) the output must be in the range of the values proposed by non-faulty processes, and (3) the difference in output between any two non-faulty processes in bounded by a constant. ABA is over a scalar value, but what about the case of having a vector quantity that we wish to approximately agree upon? The natural extension to the second requirement of ABA, that output values be in the range of input values from non-faulty processes, is that output values be in the convex hull of input values from non-faulty processes. This turns out to be much more difficult to achieve than scalar approximate agreement and the difficult appears to increase exponentially with dimension. The 1-dimensional case is a special case of the multi-dimensional case and has the same complexity.

The Multi-dimensional Approximate Byzantine Agreement (MDABA) [39, 47, 48] problem is approximately agreeing upon a vector where every process proposes a vector and up to t processes can be arbitrarily faulty. It is required that the final vector at all non-faulty processes be within the convex hull of the proposed vectors of non-faulty processes. Performing approximate

agreement on each dimension separately does not satisfy the convex hull criteria. This can be important to some problems as satisfying convexity means the output is valid. For example, if the vectors represented empirically measured probability distributions, then satisfying the convex constraint means that the output is a convex combination of the non-faulty measurements and is a valid probability distribution. Chapter 4 presents an equivalence between a piece of the MDABA and the center point from combinatorial geometry.

1.2.4 Reducing Communication Complexity in All-to-All Broadcasts

What does it mean to know something? In a distributed or concurrent system, each node has things the actor knows. But then, there are also things that are known that others know. And this can be layered, such as "I know that you know that I know something." Many distributed problems can be classified by what level of knowledge they require [28]. For problems that require higher levels of knowing, every node may need to send to every node what it currently knows. Naively done, this results in $O(mn^2)$ messages bit complexity. One solution is to make a spanning tree and then only communicate over that spanning tree. For example, elect a leader, everyone sends what they know to the leader, the leader collects all information and sends to everyone. But quiet a lot of information is duplicated in even such a simple broadcast. Chapter 5 considers the approach of using systematic error correction codes to allow the nodes to only communicate the differences in what they know. An example using gradecast is given. The gradecast algorithm, first proposed by Feldman

and Micali [19], is a broadcast algorithm that gives the receivers a confidence level in the value received. The confidence level returned is from the set $\{0, 1, 2\}$ and the confidence value gives information about the state of the other processes. The gradecast algorithm provides three main properties of the confidence level that allow a process to reason about the knowledge of other processes.

1. For all non-faulty process P_i, and non-faulty process P_j, and any process P_k, if $\mathit{confidence}_j[k] > 0$ and $\mathit{confidence}_i[k] > 0$; then, $\mathit{value}_j[k] = \mathit{value}_i[k]$.

2. For any non-faulty process P_i, and non-faulty process P_j, and any process P_k, $|\mathit{confidence}_i[k] - \mathit{confidence}_j[k]| \leq 1$.

3. If P_k is non-faulty, then for all non-faulty processes P_i, $\mathit{confidence}_i[k] = 2$ and $\mathit{value}_i[k] = v_k$.

The original one-to-all gradecast algorithm broadcasts a value from one process to all the other processes. Message bit complexity is defined as the total number of bits sent by all non-faulty processes in one invocation of the algorithm. The one-to-all gradecast algorithm has a message bit complexity of $O(mn^2)$, where m is the length of the message and n is the number of processes. The properties of gradecast make it a useful primitive in distributed systems.

Consider the case where all processes wish to broadcast a value to all other processes using gradecast. This is referred to as all-to-all gradecast and

it is used in many applications such as Byzantine agreement, approximate agreement, and multiconsensus [3]. The standard implementation of all-to-all gradecast, where n instances of the one-to-all gradecast algorithm are used, has $O(mn^3)$ message bit complexity. Chapter 5 shows a method, using coding, that gives an all-to-all gradecast algorithm with only $O(mtn^2)$ message bit complexity, where t is the specified maximum number of faulty processes. [7,8] This is a significant reduction in message bit complexity when t is much smaller than n, which is usually the case. Furthermore, gradecast requires $t < n/3$ for correctness.

1.2.5 Error Correction Codes to Synchronize Memory Access

In a concurrent program, reading and writing values can cause conflict if a read happens during a write or if more than one writer writes at the same time. Many programs require data structures to satisfy certain validity constraints. If a data structure is in the middle of being modified and someone reads it, they can't know what was modified and what was not modified. So, some sort of synchronization is required. Chapter 6 presents a method to achieve atomic updates of a buffer using error correction codes for synchronization.

The method is basically as follows. Model a buffer of memory as a point in a vector space. Each write is like moving this point by adding a vector. Some modifications wish to be relative, and some with to be absolute. For reasoning about synchronization, it is wanted that each update appear

atomic to observers. But we know that memory subsystems have a maximum sized word that they can modify at a time, so the systems notion of time must pass between updates of elements of this point. How then can the point be read in such a way that it appears to move atomically from updates? Error correction codes map a low dimension space into a higher dimension space with good separation. Pick such a mapping for a point representing the memory such that all updates that are going to be performed do not move the point in the higher space more than half the minimum distance, and a line between any two code points is only ever closest to those two points. A writer then incrementally moves the point from the old code to the new code. Now, using this code, it is possible for observers to capture a snapshot during an update that will either be the new value or the old value.

1.3 Overview of Dissertation

The rest of the dissertation is laid out as follows. First, in Chapter 2, the idea of Weighted Byzantine Agreement is given. Chapter 3 discusses a method track faults with weights in a system performing repeated Byzantine Agreement. Chapter 4 gives a result relating approximate Byzantine agreement of vectors to combinatorial geometry. Then, Chapter 5 shows the usage of error correction codes to reduce message complexity in a distributed algorithm. Finally, Chapter 6 presents a method to use error correction codes to perform atomic memory operations in a concurrent shared memory system.

Chapter 2

Byzantine Agreement

2.1 Introduction

This chapter defines a weighted version of the Byzantine Agreement Problem (WBA) [25] and provides lower bounds and algorithms for it. In WBA, each process P_i is assigned a weight $w[i]$, such that $0 \leq w[i] \leq 1$ and the sum of all weights is 1. The WBA problem requires a protocol to reach consensus when the total weight of the failed processes is at most ρ. The weighted version gives some surprising results for the BA problem. First, even if greater than $N/3$ processes are Byzantine, the system can still reach consensus so long as ρ is less than $1/3$. This result is quite useful in the system with a small set of trusted processes and a large set of less trusted processes.

Secondly, the message complexity and the number of rounds required to achieve consensus for the weighted version is shown to always be less than or equal to those for the unweighted version. Suppose the system must tolerate $\rho = f/N$ for any integer f such that $0 \leq f < N/3$. It is known that any

---

The work presented in this chapter is based on the following publication.

Vijay K. Garg and John Bridgman. The weighted byzantine agreement problem. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings*, pages 524–531. IEEE, 2011.

protocol for BA requires at least $f + 1$ rounds [16]. The unweighted version of the Queen algorithm [5] requires $f + 1$ rounds, each with two phases. In this chapter, the *anchor* of a system (denoted by α_ρ) is defined as the least number of processes whose total weight exceeds ρ. This chapter shows that WBA can be solved with the number of rounds equal to the system's anchor. The anchor for a system with $\rho = f/N$ is $f + 1$ at most and, in many cases, is much smaller than $f + 1$. Two algorithms for the WBA problem are given: the weighted-Queen algorithm and the weighted-King algorithm. These algorithms are generalizations of the algorithms proposed by Berman and Garay [5] and Berman, Garay and Perry [4]. The weighted-Queen algorithm takes α_ρ rounds, each with two phases, and can tolerate any combination of failures so long as $\rho < 1/4$. The weighted-King algorithm takes α_ρ rounds, each with three phases, and can tolerate any combination of failures so long as $\rho < 1/3$. The weighted version of BA gives a general framework to study many algorithms by instantiating BA with different weights. When the weight vector is $(1, 0, 0, 0..0)$, our algorithm reduces to a centralized algorithm, where the first process is expected not to fail and any number of other processes may fail. If M out of N processes are considered more trusted, two classes of processes can be specified by setting the weight of the M trusted processes to $1/M$ and 0 for rest. The traditional BA problem is represented by setting all the weights to $1/N$.

A general approach to the problem of knowing structure to the ways processes can fail, has been considered by Hirt and Maurer [30]. Hirt and

Maurer come up with what is called an adversarial structure. This structure is the set of all subsets of processes whose failure must be tolerated. Consider a set of processes $P = \{d, e, f, g, h, i\}$; then, the adversarial structure A would be all subsets of P that should be tolerated. Hirt and Maurer show that as long as the union of any three sets in the adversarial structure does not contain all processes in P; then, an algorithm for Byzantine agreement that can tolerate that adversarial structure exists. The basis of A, which is all maximal sets in A, is $\bar{A}$.

$$\bar{A} = \{\{d, e, f\}, \{d, g\}, \{e, h\}, \{e, i\}, \{f, g\}\}$$

The traditional BA algorithm can only handle one faulty process from P; but, there exists an algorithm that can handle at least one additional faulty process for any one faulty process as indicated by $\bar{A}$. Fitzi and Maurer [22] give an algorithm that can perform Byzantine agreement on such an adversarial structure. The algorithm by Fitzi and Maurer has message complexity polynomial in the number of processes and round complexity of less than two times the number of processes. The round complexity for our algorithms are optimal. For this particular example, a weight assignment can be found so that the King algorithm presented here can tolerate all processes in one of the subsets of P in A being faulty. One such weight assignment is shown in Table 2.1. Also, the adversarial structure may be exponentially larger than the weight assignment. The algorithms presented in this chapter are considerably simpler than the one presented by Fitzi and Maurer.

Others have considered the use of artificial neural networks (ANN) for

Table 2.1: Weight assignments for P which satisfy the adversarial structure A.

Process	d	e	f	g	h	i
Weight	1/9	1/18	8/57	1/6	5/19	5/19

BA [36, 49]. The ideas in this chapter could also be extended to use ANN. Randomization or authentication is not assumed to be available. There are many algorithms for BA with randomization [6, 43] or with authentication [16,42]. The method for using and updating weights presented here is expected to be applicable in these settings as well.

Methods to update weights for future rounds of WBA are also discussed. The weight update method guarantees that the weight of a correct process is never reduced. When the maximum weight of failure is less than 1/4, a faulty process suspected by correct processes whose total weight is at least 1/4 will be reduced to 0. Initial weight assignment is application specific. Some ideas for weight assignment and resulting probabilities and round complexity are discussed. The WBA algorithm will fail if $\rho \geq 1/3$ for any round. $\rho \geq 1/3$ is equivalent to $N/3$ failed processes in traditional BA.

The organization of the rest of this chapter is as follows. First, the model of the system that runs algorithms is defined. Second, the Weighted-Queen Algorithm is given. Next, the Weighted-King Algorithm is discussed. Following that, a simple weight update method is given. Then, some initial weight assignment strategies are presented. Finally, concluding remarks are made.

2.2 WBA Problem Specification

This chapter assumed the synchronous message passing model described in Chapter 1 Section 1.1. The system model used in this chapter is a distributed system with N processes, $P_1..P_N$, with a completely connected topology. The processes may fail in an arbitrary fashion; in particular, they may lie and collude with other failed processes to foil any protocol. The processes that do not fail in any computation are called *correct* processes. Assume that there is a non-negative weight $w[i]$ associated with each process P_i. All processes in the system have complete knowledge of weights of all the processes. For simplicity, assume that weights are normalized; i.e., the sum of all weights is one. Let ρ be the sum of weights of all failed processes. This chapter assumes that ρ is strictly less than one.

The Weighted Byzantine Agreement (WBA) problem can be specified as follows. All processes propose a binary value with the goal of deciding on one common value. Given a weight assignment to all processes, and the assumption that the weight of the processes that fail during the execution is at most ρ, the WBA problem is to design a protocol that satisfies the following conditions:

- **Agreement**: Two correct processes cannot decide on different values.
- **Validity**: The value decided must be proposed by some correct process.
- **Termination**: All correct processes decide in finite number of steps.

The following lower bounds follow easily from the standard BA lower bound arguments.

Lemma 1. *There is no protocol to solve the WBA problem for all values of w when $\rho \geq 1/3$.*

Proof. Any protocol to solve WBA can be used to solve standard BA by setting $w[i] = 1/N$ for all i. For this weight assignment, $\rho \geq 1/3$ implies that the number of failed processes f in the standard BA protocol is at least $N/3$. It is well-known that no protocol exists for standard BA when $3f \geq N$ [42]. □

For simplicity, also assume that weights associated with P_i are in non-increasing order. This can be achieved by renumbering processes, if necessary. Given any ρ and weight assignment w, define the *anchor* α_ρ as the minimum number of processes such that the sum of their weights is strictly greater than ρ. Formally,

$$\alpha_\rho = min\ \{k \mid \sum_{i=1}^{i=k} w[i] > \rho\}.$$

To get an insight into α_ρ, consider the case when ρ is f/N and each process has equal weight $1/N$. In this case, α_ρ equals $f+1$. The significance of α_ρ is that at least one process from $P_1..P_{\alpha_\rho}$ is guaranteed to be correct. When ρ is zero, α_ρ is 1. The largest possible value of α_ρ is N, because $\rho < 1$ by assumption. The following lower bound on the number of rounds for any consensus protocol is obtained from standard consensus arguments.

Lemma 2. *Any protocol to solve the WBA problem for a system with $\rho < 1$ takes at least α_ρ rounds of messages, in the worst case.*

Proof. If not, a protocol exists to solve BA in less than $f + 1$ rounds when all weights are uniform. □

We assume that communicate is synchronous and we posses a broadcast as shown in Algorithm 1.

Algorithm 1 Communicate Procedure

```
1: function COMMUNICATE(i, V, w)
2:     if w[i] > 0 then
3:         BROADCAST(V[i])
4:     end if
5:     for j such that w[j] > 0 do
6:         V[j] ← RECEIVE(j)
7:     end for
8: end function
```

2.3 Weighted-Queen Algorithm

In this section, an algorithm is given that takes α_ρ rounds, each round of two phases, to solve the WBA problem. The algorithm is based on the unweighted version of the algorithm given by Berman and Garay [5]. The algorithm uses constant-size messages, but requires that $\rho < 1/4$. Each process has a preference for each round, which is initially its input value. The algorithm shown in Figure 2 is based on the idea of a rotating queen (or coordinator). Processor P_i is assumed to be the queen for round i. In the first phase of a round, each process exchanges its value with all other processes. Based on the values received and the weights of the processes sending these values, the process determines its estimate in the variable *myvalue*. In the second phase,

the process receives the value from the queen. If P_i receives no value (because the queen has failed), then P_i assumes 0 (a default value) for the queen value. Now, P_i decides whether to use its own value or the *queenvalue*. This decision is based on the sum of the weights of the processes which proposed *myvalue* given by the variable *myweight*. If *myweight* is greater than 3/4, then *myvalue* is chosen for $V[i]$; otherwise, *queenvalue* is used.

Algorithm 2 Queen Algorithm for Weighted Byzantine Agreement at P_i

```
1:  function WEIGHTED-QUEEN-BA(i, V_i, w[N])
2:      V ← array(N) — N length array initialized to 0
3:      V[i] ← V_i
4:      for q = 0 to α_ρ do
                                                          ▷ First Phase
5:          COMMUNICATE(i, V, w)
6:          myweight ← Σ_{j|V_j=1} w[j]
7:          myvalue ← 1
8:          if myweight ≤ 1/2 then
9:              myweight ← Σ_{j|V_j=0} w[j]
10:             myvalue ← 0
11:         end if
                                                          ▷ Second Phase
12:         if q = i then
13:             BROADCAST(myvalue)
14:         end if
15:         queenvalue ← RECEIVE(q)
16:         if myweight > 3/4 then
17:             V[i] ← myvalue
18:         else
19:             V[i] ← queenvalue
20:         end if
21:     end for
22:     return V[i]
23: end function
```

The correctness of the protocol is shown by the following sequence of lemmas.

Lemma 3 (Persistence of Agreement)**.** *Assuming $\rho < 1/4$, if all correct processes prefer a value v at the beginning of a round; then, they continue to do so at the end of the round.*

Proof. If all correct processes prefer v, then the value of *myweight* for all correct processes will at least be $3/4$; because, ρ is at most $1/4$. Hence, they will choose *myvalue* in the second phase and ignore the value sent by the queen. □

Lemma 4. *There is at least one round in which the queen is correct.*

Proof. By assumption, the total weight of processes that have failed is ρ. The for loop is executed α_ρ times. By definition of α_ρ, there exists at least one round in which the queen is correct. □

Now the correctness of the protocol can be shown.

Theorem 1. *The algorithm in Figure 2 solves the agreement problem for all $\rho < 1/4$.*

Proof. The validity property follows from the persistence of agreement. If all processes start with the same value v, then, v is the value decided. Termination is obvious because the algorithm takes a fixed number of rounds. Next, the agreement property is shown. From Lemma 4, at least one of the rounds

has a correct queen. Each correct process decides either on the value sent by the queen in that round or its own value. It chooses its own value w only if *myweight* is at least 3/4. Therefore, the queen of that round must have weight of at least 1/2 for that value; because, at most, 1/4 of the weight in P_i is from faulty processes. Thus, the value chosen by the queen is also w. Hence, each process decides on the same value at the end of a round in which the queen is non-faulty. From persistence of agreement, the agreement property at the end of the algorithm follows. □

Let us analyze the algorithm's message complexity. There are α_ρ rounds, each with two phases. In the first phase, all processes with positive weight send messages to all processes. This phase results in pN messages where $p \leq N$ is the number of processes with positive weight. The second phase uses only N messages. Thus, the total number of messages is $\alpha_\rho(pN + N)$. The number of messages can be further reduced by sending messages to zero weight processes only in the last round. Note that the algorithm from [5] takes $f + 1$ rounds (each with two phases) when the maximum number of allowed failures is f. The following lemma shows that the number of rounds for the weighted version is at most the number required for the unweighted version.

Lemma 5. *$\alpha_{f/N} \leq f + 1$ for all w and f.*

Proof. It is sufficient to show that for all f,

$$\sum_{i=1}^{i=f} w[i] \geq f/N.$$

Suppose $\sum_{i=1}^{i=f} w[i] < f/N$ for some f. This implies that the sum of the remaining weights is $\sum_{i=f+1}^{i=n} w[i] > (N-f)/N$, because all weights add up to 1. Since w is in non-decreasing order, $w[f+1] > 1/N$; otherwise, the sum of the remaining weights would be at most $(N-f)/N$. But, this implies that $\sum_{i=1}^{i=f} w[i] > f/N$, because $w[i]$ for all $i \leq f$ is at least $w[f+1]$. This contradicts our original assumption. □

2.4 Weighted-King Algorithm

This section gives an algorithm that takes α_ρ rounds with three phases per round to solve the WBA problem. The algorithm is based on the *Phase King* algorithm by Berman, Garay and Perry [4]. The King algorithm only requires $\rho < 1/3$; but, adds an additional phase per round compared to the Queen algorithm. The King algorithm is given in Figure 3. As in the Queen algorithm, the King algorithm has a rotating coordinator. It is assumed that the coordinator for round k is process P_k. Each process P_i has a current preference $V[i]$ which can be 0, 1, or *undecided*. Initially, for every P_i, $V[i]$ is either 0 or 1.

In the first phase, if process P_i has a positive weight, then P_i sends $V[i]$ to all processes including itself and receives values from other processes. Next, if the cumulative sum of the weights of processes that sent 0 or 1 is greater than 2/3 then P_i sets $V[i]$ to that value, otherwise P_i sets $V[i]$ to *undecided*.

In phase two, P_i first communicates its new preference of $V[i]$ to every process if P_i's weight is positive. Note that in this phase, unlike in phase one,

Algorithm 3 King Algorithm for Weighted Byzantine Agreement at P_i

1: **function** WEIGHTED-KING-BA($i, V_i, w[N]$)
2: $V \leftarrow array(N)$ – N length array initialized to *undecided*
3: $V[i] \leftarrow V_i$
4: **for** $k = 0$ **to** α_ρ **do**
▷ First Phase
5: COMMUNICATE(i, V, w)
6: **if** $\sum_{j|V[j]=0} w[j] \geq 2/3$ **then** $V[i] \leftarrow 0$
7: **else if** $\sum_{j|V[j]=1} w[j] \geq 2/3$ **then** $V[i] \leftarrow 1$
8: **else** $V[i] \leftarrow$ *undecided*
9: **end if**
▷ Second Phase
10: COMMUNICATE(i, V, w)
11: $s_0 \leftarrow \sum_{j|V[j]=0} w[j]$
12: $s_1 \leftarrow \sum_{j|V[j]=1} w[j]$
13: **if** $s_0 > 1/3$ **then** $(V[i], myweight) \leftarrow (0, s_0)$
14: **else if** $s_1 > 1/3$ **then** $(V[i], myweight) \leftarrow (1, s_1)$
15: **else** $(V[i], myweight) \leftarrow (undecided, 1 - s_0 - s_1)$
16: **end if**
▷ Third Phase
17: **if** $k = i$ **then**
18: BROADCAST($V[i]$)
19: **end if**
20: $kingvalue \leftarrow$ RECEIVE(k)
21: **if** $V[i] = undecided$ **or** $myweight < 2/3$ **then**
22: $V[i] \leftarrow kingvalue$
23: **end if**
24: **if** $V[i] = undecided$ **then**
25: $V[i] \leftarrow 1$
26: **end if**
27: **end for**
28: **return** V_i
29: **end function**

processes may propose the value *undecided*.

Then, P_i accumulates the sum of weights of processes into $s0$ for processes who propose 0, into $s1$ for processes who propose 1. The final step in phase two is for P_i to set its preference to a new value based on the cumulative weights computed in the first part of this phase. If one of the cumulative weights is greater than 1/3, P_i sets its preference to that value. If more than one of the sums is greater than 1/3, P_i gives preference to 0, then 1, then *undecided*. P_i also sets *myweight* to the cumulative weight of the value that $V[i]$ is set.

In phase three, if P_i is the king for the current phase, P_i sends its preference $V[i]$ to every process. Next, all processes receive the king's value into *kingvalue*. Then, if P_i is undecided ($V[i] = undecided$) or the weight stored in *myweight* from phase two is less than 2/3, P_i sets its preference to *kingvalue* if *kingvalue* is not *undecided* or 1 if *kingvalue* is *undecided*. After executing for α_ρ rounds, P_i outputs $V[i]$ as the decided value. The correctness of the King algorithm is shown in the following lemmas.

Lemma 6 (Persistence of Agreement)**.** *Assuming $\rho < 1/3$, if all correct processes prefer a value v at the beginning of a round; then, they continue to do so at the end of the round.*

Proof. If all correct processes agree at the beginning of the round; then, for the first phase, by definition, the same value must be chosen as $\rho < 1/3$. For the second phase, the same value must again be chosen as $\rho < 1/3$. For the third

phase, because all correct processes agree and $\rho < 1/3$, all correct processes will ignore the king's value and keep their own. □

Lemma 7. *There is at least one round in which the king is correct.*

Proof. By assumption, the total weight of processes that have failed is less than ρ. The for loop is executed α_ρ times. By definition of α_ρ, there exists at least one round in which the king is correct. □

Theorem 2. *The algorithm in Figure 3 solves the agreement problem for* $\rho < 1/3$.

Proof. Validity is satisfied by persistence of agreement. If all processes start with the same value, then that value will be decided. Termination is obvious because the algorithm takes a fixed number of rounds. From Lemma 7, in at least one round, the king will be correct. In that round, every correct process will choose either the king's value, 1, or its own value. The only way that a process may choose its own value is if *myweight* $\geq 2/3$ and the process is not undecided; otherwise, the process will choose the king's value or 1 if the king is undecided. If a process chooses its own value, then, *myweight* $\geq 2/3$ for that process and the weight of its value $V[i]$ will be $\geq 1/3$. So, the king must also have chosen the same value. If *myweight* $< 2/3$ or V is undecided, then the process will choose the kings value or 1 if the king is undecided. Because the king is correct, then all processes will choose the same value. □

The King algorithm takes α_ρ rounds with three phases per round. In phase one and two, each process with positive weight sends N messages. In phase three, the king process sends N messages. This results in $\alpha_\rho(2pN + N)$ messages where p is the number of processes with positive weight.

2.5 Updating Weights

In this section, the case when the system is required to solve BA multiple times is considered. This case arises in most real-life applications of BA, such as, maintenance of replicated data and fault-tolerant file systems [9]. In addition, each execution of the BA protocol provides certain feedback in terms of the processes' behavior. For example, if a process did not follow the protocol (i.e., did not send the required messages), it should be considered less reliable for future BA instances. In this section, a fault-tolerant method to update weights is given. For simplicity, only the weighted-Queen algorithm is given; the extension to weighted-King algorithm is similar.

The following lemma gives the conditions sufficient for P_i to detect that P_j is faulty.

Lemma 8. *In the Weighted-Queen algorithm, a correct process P_i can detect that P_j is faulty if any of the following conditions are met:*

1. *If P_j either does not send a message or sends a message with wrong format in any of the rounds, then P_j is faulty.*

2. *If myweight* $> 3/4$ *in any round and the value sent by the queen in that round is different from myvalue, then the queen is faulty.*

Proof. The first part is obvious. For the second part, note that if *myweight* $>$ $3/4$; then, the weight for the queen for that value in that round is at least $1/2$. If the queen is correct, the value sent by the queen would have matched *myvalue*. □

The algorithm in Figure 2 is modified by adding a variable *faultySet* that keeps track of all processes that P_i has detected to be faulty based on Lemma 8. Now a method is presented to update the weights of the processes such that with every execution of WBA, the processes get better in solving WBA by increasing the weights of reliable processes. These algorithms require that the weight assignment for all correct processes be identical; so, it is not sufficient for a process to update its weight individually. All correct processes need to agree on the faulty set.

The algorithm to update weights shown in Figure 4 consists of three phases. In the first phase, called the learning phase, processes broadcast their *faultySet* to learn about faulty processes from other correct processes. The main idea is that if processes with total weight at least $1/4$ inform P_i that some process P_j is faulty; then, P_j is in *faultySet* of at least one correct process. The second phase consists of processes agreeing on the set of faulty processes. For each process j, if j is in the *faultySet* of P_i , then P_i invokes

`Weighted-Queen-BA` algorithm with 1 as the proposed value; otherwise, it invokes it with 0 as the proposed value. The output variable *value* denotes the decided value by the `Weighted-Queen-BA` algorithm. Therefore, the set of faulty processes that all correct processes agree upon is *consensusFaulty*. In the third phase, processes update their weights based on *consensusFaulty*.

The correctness of the algorithm in Figure 4 is shown in the following lemma and theorem.

Lemma 9. *All correct processes with positive weights before the execution of the algorithm have identical w vectors after the execution of the algorithm.*

Proof. The weight assignment is done based on *consensusFaulty*. The variable *consensusFaulty* is identical at all correct processes based on the correctness of Weighted-Queen algorithm. □

Theorem 3. *A correct process can never be in consensusFaulty. Any faulty process that is in the initial faultySet of correct processes with total weight at least* $1/4$ *will be in consensusFaulty of all correct processes.*

Proof. A correct process P_j can never be in the initial *faultySet* of any correct process (due to Lemma 8). In the learning phase, $suspectWeight[j]$ at any process can never be equal or more than $1/4$, because only faulty processes can suspect P_j. Therefore, j is not in *faultySet* of any correct process after the learning phase. Since all correct processes will invoke WBA with 0 for P_j, by validity of the Weighted-Queen-BA algorithm, it will not be in

Algorithm 4 Weight-Update Algorithm for the Queen Algorithm for Weighted Byzantine Agreement at P_i

1: *faultySet*: set of processes based on Lemma 8
2: *consensusFaulty*: set of processes initially {}
3: *suspectWeight*: array[1..p] of float initially all 0.0
▷ First phase (learning phase)
4: **for** j **such that** $w[j] > 0$ **do**
5: send *faultySet* to all (including itself)
6: **end for**
7: **for** j **such that** $w[j] > 0$ **do**
8: receive $faultySet_j$ from P_j
9: **for** $k \in faultySet_j$ **do**
10: $suspectWeight[k] \leftarrow suspectWeight[k] + w[j]$
11: **end for**
12: **end for**
13: **for** j **such that** $w[j] > 0$ **do**
14: **if** $suspectWeight[j] \geq 1/4$ **then** $faultySet \leftarrow faultySet \cup \{j\}$
15: **end if**
16: **end for**
▷ Second phase
17: **for** j **such that** $w[j] > 0$ **do**
18: **if** $j \in faultySet$ **then** $value \leftarrow$ Weighted-Queen-BA(1)
19: **else** $value \leftarrow$ Weighted-Queen-BA(0)
20: **end if**
21: **if** $value = 1$ **then** $consensusFaulty \leftarrow consensusFaulty \cup \{j\}$
22: **end if**
23: **end for**
▷ Third phase
24: $totalWeight \leftarrow 1.0$
25: **for** $j \in consensusFaulty$ **do**
26: $totalWeight \leftarrow totalWeight - w[j]$
27: $w[j] \leftarrow 0$
28: **end for**
29: **for all** j **do**
30: $w[j] \leftarrow w[j]/totalWeight$
31: **end for**

consensusFaulty. Any faulty process that is in the initial *faultySet* of correct processes with total weight of at least 1/4 will be in *faultySet* of all correct processes after the learning phase. Again, from the validity of WBA, the faulty process will be in *consensusFaulty*. □

The model assumed here for updating weights is that once a process is faulty, it will always be faulty. A modification can be considered where a process may become non-faulty after being faulty for a period of time. In this case, instead of setting the weight to zero, the weight can be reduced by some multiple.

2.6 Weight Assignment

Deciding what weight assignment to use is application specific. A simplified example will be considered for this section. Consider two sets of processes A and B where all processes in A have probability of failure f_a and all processes in B have probability of failure f_b. We will consider four weight assignments. The first is a uniform weight assignment for everyone. This weight assignment produces the same results as the classical algorithm. The next assignment is to only give non-zero weights to the set with a lower probability of failure. The third is to give weights to each process proportional to the inverse of their probability of failure. Weights proportional to the probability of not failing is the final assignment considered.

The graph in Figure 2.1 is the probability of the weight of failed pro-

cesses exceeding 1/3 versus $|B|$ with $|A| = 6, f_a = 0.1, f_b = 0.3$. This graph is only taken for points where the number of processes is divisible by three. The number three is chosen because taking every point produces many more jumps in the graph which just add noise and distract from the trend. Notice that there are still some jumps. These jumps are caused by the effect of adding a process to a group where that addition does not increase the number of faulty processes that can be tolerated. But, adding that one process increases the expected weight of failed processes. So, there is a jump in the probability of the total weight of failed processes being above 1/3. Each curve starts at the same value because set B is empty. Observe that each curve initially has a positive average slope. It is not until a significant number of additional processes are added that the curve begins to have a negative slope. The uniform assignment gives the worse probability for a small size of B relative to the size of A. Changing the number of processes in set A moves the curves vertically in relation to each other. Which weight assignment is best depends upon the number of processes in both A and B and their probability of failure. In this particular example, setting the weight proportional to the inverse probability of failure gives the best results.

Figure 2.2 shows the number of rounds required for the King algorithm to ensure success. Notice the uniform assignment is the highest. In both of these graphs, the uniform weight assignment was the least attractive. The most attractive assignments are only giving positive weights to group A and setting the weight proportional to the inverse probability of failure. For this

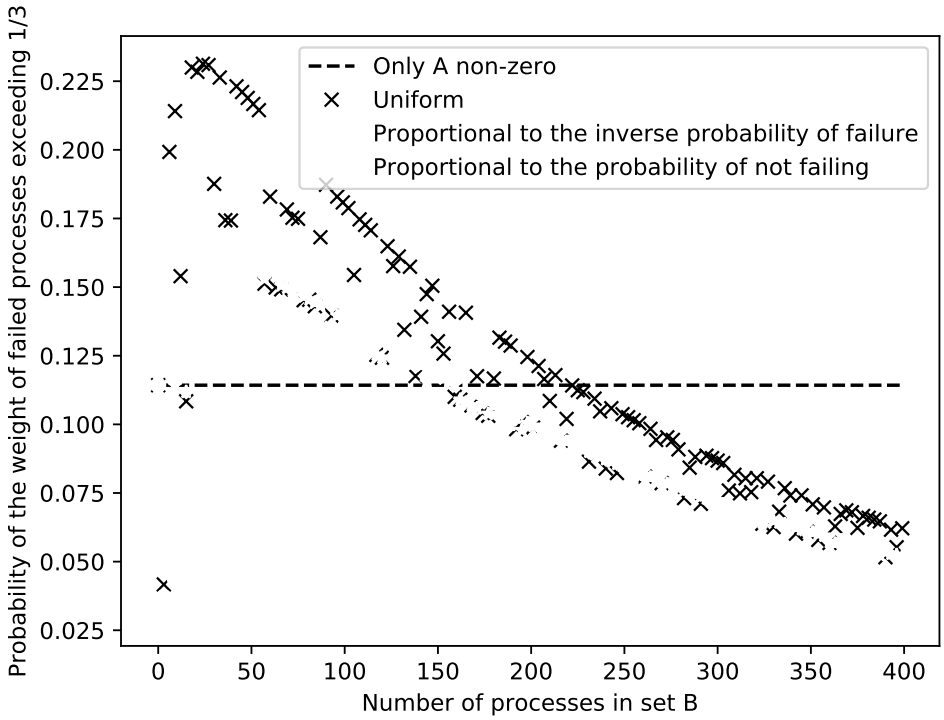


Figure 2.1: Probability of the weight of failed processes exceeding 1/3 with versus $|B|$ with $|A| = 6, f_a = 0.1, f_b = 0.3$.

particular setup, setting weights proportional to the inverse probability of failure is the best. When the size of set B is not much larger than A, only giving positive weights to set A may be the best. When the size of set B is significantly larger than A, then setting the weights to be proportional to the inverse probability of failure is the best.

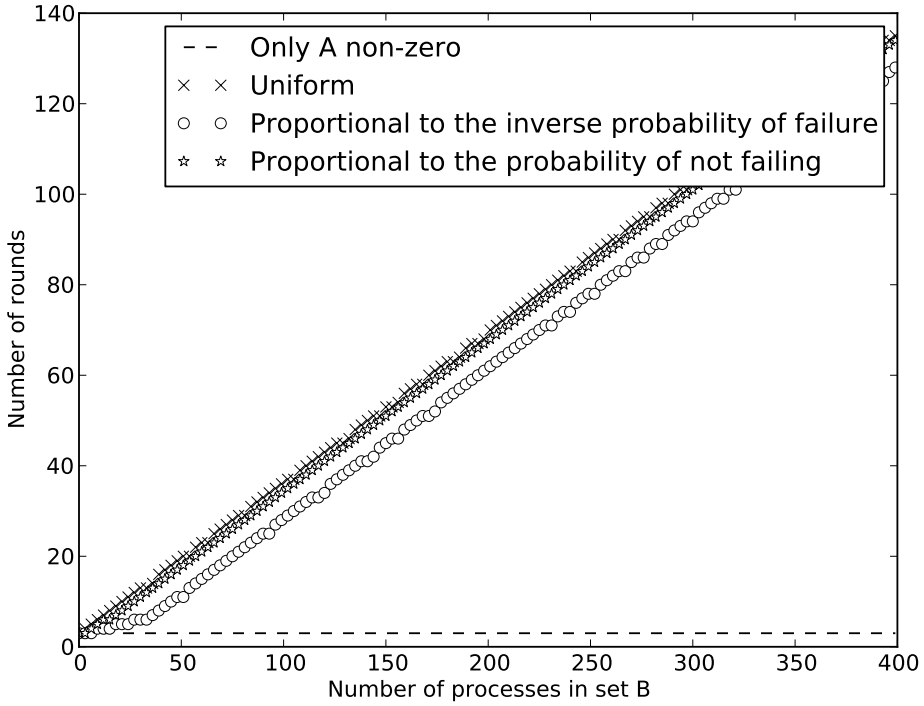


Figure 2.2: The number of rounds required for the King algorithm versus the number of processes in set B for the different weight assignments.

2.7 Conclusions

This chapter has presented a weighted version of the Byzantine Agreement Problem and provided solutions for the problem in a synchronous distributed system. We show that the weighted version has the advantage of using fewer messages and tolerating more failures (under certain conditions) than is required by the lower bound for the unweighted version. These algorithms have applications in many systems in which there are two classes of processes: trusted and untrusted processes. Instead of tolerating any f faults

in the BA problem, these algorithms tolerate failure of processes with total weight less than f/N. For example, an implementation can now tolerate more than f faults of untrusted processes; but, fewer than f faults of trusted processes depending on the weight assignment. A fault-tolerant method has also been presented to update the weights at all the correct processes. This algorithm is useful for many applications where the agreement is required multiple times. Our update algorithm guarantees that the weight of a correct process is never reduced and the weight of any faulty process, suspected by correct processes whose total weight is at least $1/4$, is reduced to 0.

Chapter 3

Accurate Byzantine Agreement with Feedback

3.1 Introduction

This chapter presents Accurate Byzantine Agreement with Feedback, a joint work with Bharath Balasubramanian and Vijay Garg [26, 27]. In the standard version of Byzantine Agreement [15, 20, 24, 35, 42], the value that is agreed upon may be either of the binary values so long as it is proposed by at least one non-faulty process. In some scenarios, it is better for the system to agree on a specific value among the two binary values. For example, suppose in a distributed control system a coordinated action needs to be taken (such as opening or closing a valve) depending upon the observations made by possibly faulty distributed processes. Depending upon the outcome of the action, the environment can provide a feedback if the action taken was correct or not. As another example, suppose that the system is making decision on whether to sell a stock based on recommendations made by multiple processes. The final closing price of the stock provides a feedback for the decision made. Thus,

---

The work presented in this chapter is based on the following publication, all authors contributed equally.

Vijay K. Garg, John Bridgman, and Bharath Balasubramanian. Accurate byzantine agreement with feedback. In Antonio Fernàndez Anta, Giuseppe Lipari, and Matthieu Roy, editors, *Principles of Distributed Systems*, pages 465–480, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

the system or the environment can usually provide feedback to the non-faulty processes about which of the values was preferred or correct for that iteration of the agreement algorithm. Can the non-faulty processes use this feedback in a way that the probability of choosing the correct value increases in subsequent iterations of the algorithm?

We refer to this version of the BA problem as Accurate Byzantine Agreement (ABA) and define it as follows. Assume a set of n processes among which at most f Byzantine faults can occur. All non-faulty processes are required to make decisions for multiple rounds or iterations. For each iteration, a process can propose a binary value 0 or 1. All non-faulty processes must agree on each decision and must take finite time to agree. After each decision, the environment provides a common feedback to all processes indicating if their decision was correct or wrong. The goal is to design an algorithm that maximizes the (expected) number of correct decisions by non-faulty processes over iterations of the algorithm.

In this chapter, we give an algorithm, referred to as the ABA algorithm for the ABA problem. Our method relies on maintaining a common weight vector at all processes and updating this vector based on the feedback for each iteration. Initially, the weight of each process is a non-negative value proportional to the trust of the system on that process. If there is no prior information available, then the weights can simply be initialized to $1/n$. We use a weighted majority rule to determine the agreed upon value for the ABA problem. Once the value is committed, the feedback determines whether the

decided value was a mistake or not. An important aspect of the algorithm is how the weights are updated based on the feedback. One possibility is to penalize all processes that proposed a wrong value after each iteration. Another possibility is to penalize processes only if the value decided in that iteration was wrong. Somewhat surprisingly, the behavior of the ABA algorithm may crucially depend upon which rule is used. We provide guarantees on the accuracy of the algorithm based on different assumptions on the accuracy of the processes and different weight update rules.

Byzantine Agreement is a well-studied problem in the field of distributed computing with research in both the theoretical [1, 22, 30, 32] and practical aspects [9, 11, 13]. For the synchronous model of communication (as assumed in this chapter), it is known that agreement can be achieved only when $n \geq 3f + 1$ [42]. In our work in Chapter 2 [25], we present algorithms and bounds for weighted BA, where processes are assigned weights according to the application. In that chapter, we give Byzantine agreement protocols that work even when $n < 3f + 1$, where f is the number of processes that have failed so long as the ratio of the weight of the failed processes to the weight of non-faulty processes is at most $1/2$. We also present techniques to increase the weights of the non-faulty processes relative to that of the faulty processes based on detection of faulty behavior. The Weighted BA problem does not have any notion of *accurate* value for agreement or environmental feedback as required for the ABA problem. It can be used as a subroutine in the ABA algorithm as shown in Section 3.5. Other approaches to BA include

the use of artificial neural networks [36, 49], randomized algorithms [6, 43] or authentication based algorithms [16, 42]. None of these works explore the notion of accurate processes or the correct value for agreement. Our work can be applied to extend the results of these papers.

The concept of weighted majority and multiplicative weight update is used in many disciplines such as learning theory, game theory and linear programming [31, 38]. In the literature for this methodology, the experts are independent entities and there is no notion of liars that can collude and confuse other experts into suggesting the wrong value. In this chapter, we assume the presence of malicious Byzantine experts and design algorithms to tolerate them. In summary, we make the following contributions:

- *The ABA Problem*: We introduce the problem of Accurate Byzantine Agreement, where the processes have to agree on a correct binary value as deemed by environmental feedback. The goal is to use this feedback to improve the accuracy of the algorithm in subsequent iterations.

- *The ABA Algorithm*: We present an algorithm to solve the ABA problem that uses a standard solution to the BA problem and a multiplicative method to maintain and update process weights. We make guarantees on the accuracy of the algorithm for the following models:

  - *Deterministic Accuracy*: We make assumptions on two ratios, the *accuracy ratio* (α) and the initial fault ratio (r_0). The accuracy ratio

is the ratio of weight of the accurate processes to the weight of the non-faulty processes. The fault ratio r is the ratio of the weight of the faulty processes to that of the non-faulty processes. When $\alpha > 3/4$, the algorithm is always accurate if $r_0 < 1/2$. We relax this bound and show that when $\alpha > (1/2 + d)$, for any $0 \leq d \leq 1/2$, the algorithm is always accurate if $r_0 < 2d$.

  - *Probabilistic Accuracy*: We make assumptions on the probability with which non-faulty processes propose the correct value, β, and on the fault ratio r. When $\beta > 1/2 + d$ for any $0 < d < 1/2$, the probability of the algorithm being inaccurate is exponentially small if $r < 2d$.

  - *At-Least-One Accuracy*: If there exists at least one process such that it is inaccurate at most b times, then the ABA algorithm is inaccurate only $O(b + \log n)$ times. Hence, the algorithm tracks the most accurate process in the system.

- *Experimental Evaluation*: We present simulation results evaluating the performance of three distinct solutions: the ABA algorithm (with update on inaccuracy), the ABA algorithm with update on every iteration (always update) and the standard Byzantine Agreement (never update). While always-update and never-update perform very well for one of the models each, they perform poorly for the other one. The update on inaccuracy method performs well for both the models.

3.2 Model and Definitions

This chapter assumes the execution model described in Chapter 1 Section 1.1. Furthermore, we classify the processes in our system based on their behavior into non-faulty, accurate and faulty processes. While the notion of faulty and non-faulty processes is common to all BA problems, we introduce the concept of accurate processes that captures the idea of a correct proposal. A non-faulty process is considered *accurate* for an iteration if it proposes the correct value for that iteration.

In the standard BA problem, all non-faulty processes must agree on a common value. The only requirement on the decided value is that it must be proposed by a non-faulty process. In our proposal, the value decided by the algorithm is important as there is a reward function associated with the value decided, awarded by the environment or the system. The *correct* value is assigned 1 unit of reward and an incorrect value is assigned 0 units, i.e., no reward. Based on the reward, we replace the standard concept of validity with the notion of *accuracy*. Validity specifies that the value decided by the non-faulty processes must have been proposed by at least one of the non-faulty processes. This condition eliminates the trivial solution where all non-faulty processes agree on a fixed value all the time. In our system, the accuracy requirement eliminates the trivial solution. We define our problem below.

Definition 1. *(Accurate Byzantine Agreement with Feedback) Consider n processes consisting of non-faulty and faulty processes. There are multiple binary*

decisions that these n processes are required to make. For each possible decision (iteration of the ABA problem), each of the non-faulty processes proposes either 0 *or* 1*. An algorithm that solves the Accurate Byzantine Agreement with Feedback (ABA) problem, must guarantee the following properties:*

- *Agreement: For each iteration, all non-faulty processes decide on the same value.*
- *Termination: The algorithm terminates in a finite number of rounds.*
- *Accuracy: The non-faulty processes agree on a value that is deemed correct by environmental feedback.*

To incorporate the feedback provided by the environment we assign a non-negative weight w_i to each process P_i that provides an estimate, possibly erroneous, of the trust placed on that process. We summarize our notation in Table 3.1.

Table 3.1: Notation in Chapter 3

n	Number of processes	f	Number of Byzantine faults
w_i	Weight of process P_i	a	Total weight of accurate processes
p	Total weight of non-faulty processes	q	Total weight of faulty processes
r	Fault Ratio ($= q/p$)	α	Accuracy ratio ($= a/p$)

3.3 The ABA Algorithm

In this section, we propose an algorithm (Algorithm 5) for the ABA problem. The algorithm is identical at all processes and executes in syn-

Algorithm 5 The ABA Algorithm at P_i

$W \leftarrow array(N)$ initialized to system trust (default value all $1/n$)
$V \leftarrow array(N)$ initialized to 0
$t \leftarrow$ total number of iterations
for *iteration* $= 1$ to t **do**
    $V[i] \leftarrow$ value proposed by P_i
                                ▷ Step 1: Exchange values with all
    BROADCAST($V[i]$)
    for $j = 1$ **to** N **do**
        $V[j] \leftarrow$ RECEIVE(j)
    end for
                                ▷ Step 2: Agree on V
    for $j = 1$ **to** N **do**
        $V[j] \leftarrow$ BYZANTINEAGREEMENT($V[j]$)
    end for
                                ▷ Step 3: Compute support and choose majority
    $decided \leftarrow \sum_{\{j:V[j]=0\}} W[j] < \sum_{\{j:V[j]=1\}} W[j]$
    ▷ Step 4: Wait for reward and determine the correct value based on the feedback
    $correctVal \leftarrow reward = 1 \ ?\ decided : \neg decided$
        ▷ Step 5: multiplicative weight update on inaccuracy: ABA(UI)
    if $reward = 0$ **then**
        for $j : V[j] \neq correctValue$ **do**
            $W[j] \leftarrow W[j] \cdot (1 - \epsilon)$
        end for
    end if
    ▷ Alternative Step 5': multiplicative weight update on all iterations ABA(UA)
    for $j : V[j] \neq correctValue$ **do**
        $W[j] \leftarrow W[j] \cdot (1 - \epsilon)$
    end for
end for

chronous iterations. At each process, we maintain two vectors W and V. Vector W stores the weight of each process while vector V stores the value proposed by each of them. Initially, the weight of each process is a non-negative value directly proportional to the initial trust on that process. In each iteration of the algorithm each non-faulty process proposes a value and executes Step 1 to Step 5 of the algorithm.

In Step 1, all processes exchange their proposed values to populate V. If no value is received from some process, the corresponding entry is set to 0. Since faulty processes may send conflicting values to other processes, it is not guaranteed that the V vector is identical at all non-faulty processes after Step 1.

In Step 2, the algorithm requires all non-faulty processes to agree on the value proposed by every other process and thereby make the V vector identical at all non-faulty processes after Step 2 of any iteration. For this step, we can use any standard BA algorithm such as the King algorithm [4] that requires $n \geq 3f + 1$, or the Queen algorithm [5] that requires $n \geq 4f + 1$. The validity property satisfied by these algorithms ensures that the value of $V[i]$ for any non-faulty process P_i is exactly the value proposed by P_i.

In Step 3, processes determine the sum of weights of all processes that support value 0 or 1. The value with larger support, i.e., the weighted majority is chosen as the value in *decided*.

In Step 4, processes receive the common feedback from the environment

to determine the correct value.

In Step 5, we carry out the update of weights. If the value decided was incorrect, then the weights of the processes that proposed an incorrect value is reduced by some constant proportion ϵ ($0 < \epsilon < 1$) of its previous weight (multiplicative update). As an alternative to step 5, in step 5', we carry out the weight update on all iterations irrespective of the reward value. If we update weights only on inaccuracy, we refer to the algorithm as ABA(UI) ("update on inaccuracy"). If we update weights on all iterations, we refer to the algorithm as ABA(UA) ("update always"). We now prove that both the versions of the algorithm guarantee the agreement and termination property specified in Definition 1 independent of the assumptions on accuracy.

Theorem 4. *(Agreement & Termination) Assuming $n \geq 3f+1$, all iterations of the ABA algorithm guarantee agreement and termination.*

Proof. Agreement: We show that after Step 2 of every iteration, all non-faulty processes have identical W and V vectors. The proof is by induction on the iteration number. At the first iteration, the vector W is identical at all non-faulty processes by the initialization. Now, assume that the vector W is identical at the beginning of any iteration i. Because all processes agree on vector V using Byzantine agreement, all non-faulty processes will have identical V after Step 2. This implies that all non-faulty processes will have identical support for 0, support for 1 and value of *decided* after step 3 because these variables depend only on W and V. Since the reward function is assumed to

be common, all non-faulty processes will have identical value of *correctVal* and therefore will update W in an identical manner. The value decided depends only on W and V vectors and hence all non-faulty processes agree on the same value.

Termination: This is a synchronous algorithm which executes in finite number of rounds and hence, termination is satisfied trivially. □

The ABA algorithm guarantees another useful property: if a non-faulty process proposes an accurate value, then it can never be penalized. This property exploits the validity condition satisfied by the BA algorithm used in Step 2. A non-faulty process P_i will send the same value to all non-faulty processes. Therefore, all non-faulty processes will have identical $V[i]$ when they invoke the BA algorithm. Therefore, by validity of the BA algorithm, $V[i]$ at all non-faulty processes will be identical to the one proposed by P_i.

3.4 Accuracy Guarantees of the ABA Algorithm

In the previous section, we have shown that ABA algorithm guarantees agreement and termination. This section focuses on the accuracy guarantees the algorithm can provide based on varying assumptions about the accuracy of the processes in the system. Since standard Byzantine agreement is used in Step 2, in this section we assume that $n \geq 3f + 1$, according to the lower bound for the BA problem [42]. In Section 3.5, we consider the case when $n \geq 3f + 1$ does not hold.

3.4.1 Deterministic Accuracy

For deterministic accuracy, we make guarantees based on the accuracy ratio α (ratio of the weight of accurate processes to the weight of non-faulty processes) and the fault ratio of the system r (ratio of the weight of faulty processes to the weight of non-faulty processes). We show that if $\alpha > 3/4$ for each iteration and if the initial fault ratio $r_0 < 1/2$; then, the algorithm guarantees accuracy. Next, we relax this requirement and show that it is sufficient that $\alpha > (1/2 + d)$ for each iteration such that $r_0 < 2d$, to guarantee accuracy.

We first show that as long as $\alpha > 1/2$ for each iteration, r never increases if we update weights only on error. This enables us to make guarantees just based on the initial fault ratio of the system. The proof crucially depends on the fact that we update the weights of inaccurate processes only when the algorithm chooses the incorrect value.

Lemma 10. *(Non-Increasing Fault Ratio) For any iteration, if the accuracy ratio $\alpha > 1/2$, then the fault ratio r cannot increase after that iteration of the ABA(UI) algorithm.*

Proof. In the ABA(UI) algorithm, the weights of the processes changes only when the algorithm makes a mistake. Consider the weight of the non-faulty processes, p. Since $\alpha > 1/2$, when the algorithm makes a mistake, greater than $p/2$ of the weight will be unaffected and less than $p/2$ of the weight will be reduced by a factor of $1 - \epsilon$. Hence, if p' is the weight of the non-faulty

processes after a weight update,

$$p' > p/2 + (1-\epsilon)p/2 = p(2-\epsilon)/2. \tag{3.1}$$

Now, consider the weight of the faulty processes q. The algorithm chooses the wrong value only when a majority weight, i.e. $> (p+q)/2$ of the weights are inaccurate. Since greater than $p/2$ of the weights are accurate, at least $q/2$ of the weights are inaccurate. Hence, if q' is the weight of the faulty processes after a weight update,

$$q' < q/2 + (1-\epsilon)q/2 = q(2-\epsilon)/2. \tag{3.2}$$

Dividing equation 3.2 by equation 3.1, we get, $q'/p' < q/p$. □

Note that the proof for Lemma 10 does not hold for the always update rule. If the faulty processes keep proposing the correct value, then the ABA(UA) algorithm will increase the relative weight of the faulty processes and consequently the fault ratio. If the fault ratio increases beyond 1, then Byzantine processes can force the ABA algorithm to choose incorrect values on crucial decisions.

In the following theorem we show that if $\alpha > 3/4$, then the ABA(UI) algorithm never makes a mistake as long as the initial fault ratio is less than $1/2$.

Lemma 11. *If the accuracy ratio $\alpha > 3/4$ for all iterations, and the initial fault ratio $r_0 < 1/2$, then the ABA(UI) algorithm always guarantees accuracy.*

Proof. If the accuracy ratio a/p is greater than $3/4$, then the weight of accurate proposals a is at least $3p/4$. This implies that the weight of inaccurate proposals is at most $p + q - 3p/4 = p/4 + q$. The algorithm selects the correct value if the accurate weight is more than the inaccurate weight. We need to show that, $p/4 + q < 3p/4$. Dividing both sides by p and rearranging, this is equivalent to showing that $r < 1/2$. Since $r_0 < 1/2$, from Lemma 10, for all iterations, $r < 1/2$. Note that, for the ABA(UI) algorithm, we update the weights only when the algorithm makes a mistake. So for any iteration, if $\alpha > 3/4$ and $r < 1/2$, it will remain so for every subsequent iteration and hence the ABA(UI) algorithm never makes a mistake. □

In the following theorem, we show that even if the accuracy ratio is just above $1/2$, the ABA algorithm never makes a mistake as long as the initial fault ratio is less than a certain threshold.

Theorem 5. *(Deterministic Accuracy) If the accuracy ratio $\alpha > 1/2 + d$ for all iterations and if the initial fault ratio $r_0 < 2d$, for any $0 \leq d \leq 1/4$, then the ABA(UI) algorithm always guarantees accuracy.*

Proof. If the weight of accurate proposals is at least $p(1/2 + d)$, then the weight of inaccurate proposals is at most $p(1/2 - d) + q$. The algorithm selects the correct value if the accurate weight is more than the inaccurate weight. Therefore, we need $p(1/2 - d) + q < p(1/2 + d)$. This condition is equivalent to $r < 2d$. Since $r_0 < 2d$, from lemma 10, for any iteration, $r < 2d$. Since

the correct decision was made, the weights are not updated and the algorithm continues to chose the correct value in the subsequent iterations. □

Note that when d equals $1/4$, this theorem reduces to lemma 11. Thus, theorem 5 generalizes lemma 11, when $d < 1/4$. Accuracy of the ABA(UI) is guaranteed if either an overwhelming majority of non-faulty processes is accurate (d is large) or there is a large percentage of non-faulty processes (r_0 is small).

In the following theorem, we make guarantees based on the number of accurate processes and the number of faulty processes in the system.

Theorem 6. *If the number of accurate processes is greater than* $1/2 + d$ *times the number of non-faulty processes for all iterations and if the initial number of faulty processes is less than* $2d$ *times the number of non-faulty processes, for any* $0 \leq d \leq 1/4$*, then the ABA(UI) algorithm always guarantees accuracy.*

Proof. We initialize the weights of all processes to $1/n$. This proof follows directly from theorem 5. If the number of accurate processes is greater than $1/2 + d$ times the number of non-faulty processes then the accuracy ratio $\alpha > 1/2 + d$, since the weights are equally initialized. Similarly, the initial fault ratio $r_0 < 2d$. Hence from theorem 5, the ABA(UI) algorithm guarantees always guarantees accuracy. As mentioned in the proof of theorem 5, since the algorithm decides on the correct value, the weights are not updated and hence the algorithm continues to chose the correct value in the subsequent iterations. □

The following theorem handles the case when a majority of the non-faulty processes are accurate but the fault ratio is not smaller than $2d$.

Theorem 7. *(Accuracy after some initial mistakes) If the accuracy ratio $\alpha > 1/2 + d$ for all iterations, for any $0 \leq d \leq 1/4$, then the ABA(UI) algorithm guarantees accuracy after some initial mistakes.*

Proof. (Sketch) Similar to the proof of lemma 10, it is easy to show that there exists a constant γ such that the fault ratio decreases by a factor of at least γ for any mistake. Therefore, eventually the fault ratio becomes less than $2d$. Subsequently, by theorem 5 the algorithm ABA(UI) does not make any mistake. □

It is easy to show that ABA(UA) algorithm can be forced to make unbounded mistakes by the Byzantine processes for any accuracy ratio less than 3/4. Byzantine processes may initially propose correct values to increase the fault ratio. Once the fault ratio is high, they can ensure that ABA makes mistakes. They can repeat this cycle forever.

3.4.2 Probabilistic Accuracy

For probabilistic accuracy, we make guarantees based on the probability of accuracy of each non-faulty process β, and the fault ratio r, of the system. We show that if $\beta > 1/2 + d$ $(0 < d < 1/2)$, and $r < 2d$, the ABA algorithm guarantees accuracy with high probability.

Theorem 8. *(Probabilistic Accuracy) Let all weights in the system be in* $[0, 1]$. *If the accuracy probability of non-faulty processes* $\beta > 1/2 + d$ *and the fault ratio* $r < 2d$, *for any* $(0 < d < 1/2)$, *for all iterations, then the ABA algorithm guarantees accuracy with probability greater than* $1 - (\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}})^{\mu}$, *where* $\mu = p(1/2 + d)$ *and* $\delta = (2d - r)/(2d + 1)$.

Proof. Let X_i be the random variable indicating the non-faulty process P_i making an accurate proposal. Let $X = \sum_i w_i X_i$, where w_i is the weight of process P_i. We have $E[X_i] = 1/2 + d$. Therefore, $E[X] = (1/2 + d) \sum_i w_i = p(1/2 + d)$.

Let $\mu = E[X]$. We now show that $(1 - \delta)\mu = (p + q)/2$.

$$\begin{aligned}
(1-\delta)\mu &= (2d + 1 - (2d - r))/(2d + 1) \cdot p \cdot (2d + 1)/2 \\
&= (1 + r)p/2 \\
&= (1 + q/p)p/2 \\
&= (p + q)/2.
\end{aligned}$$

When $r < 2d \leq 1$, we get that $0 < \delta < 1$. Hence, from Chernoff's bound, we have,

Pr[ABA algorithm makes wrong decision]

$$\begin{aligned}
&= Pr[\textstyle\sum_{j:V[j]=correctVal} W[j] < \frac{(p+q)}{2}] && \{\text{From the algorithm}\} \\
&\leq Pr[X < (p + q)/2)] && \{\text{Considering only non-faulty processes}\} \\
&= Pr[X < (1 - \delta)\mu] && \{\text{Shown above}\} \\
&< \left(\tfrac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^{\mu} && \{\text{From Chernoff's bound and } 0 < \delta < 1\}.
\end{aligned}$$

□

In Theorem 8, the error probability depends upon $\delta = (2d-r)/(2d+1)$. As r decreases, δ increases. We now show that for the ABA(UA) algorithm the ratio r is expected to decrease exponentially with increasing iterations.

Theorem 9. *(ABA(UA): Exponentially Decreasing Expected Fault Ratio) If the accuracy probability of non-faulty processes is at least* $1/2 + d$*, and the accuracy probability of faulty processes is at most* $1/2-d$*, then there exists* $k > 1$*, such that after* j *iterations of the ABA(UA) algorithm, the expected ratio of the weight of the non-faulty processes to the weight of the faulty processes is at least* k^j/r_0*.*

Proof. We first show a bound on the expected weight of a non-faulty process after j iterations. Let the initial weight of a non-faulty process be w_0. Let M_i be the random variable denoting the multiplicative factor at iteration i for a non-faulty process. Let W_j be the random variable denoting weight of a non-faulty process after j iterations. It is clear that for ABA(UA) algorithm, $W_j = w_0\Pi_{i=1}^{i=j}M_i$. The multiplicative factor for any iteration depends on the environmental feedback and is independent of other iterations. Hence,

$$\begin{aligned} E[W_j] &= w_0\Pi_{i=1}^{i=j}E[M_i] \\ &\geq w_0\Pi_{i=1}^{i=j}((1/2+d)\cdot 1 + (1/2-d)\cdot(1-\epsilon)) \\ &= w_0(1-\epsilon/2+d\epsilon)^j. \end{aligned}$$

Similarly, since the probability that a faulty process makes a correct proposal is at most $1/2 - d$, the expected weight of a faulty process after j iterations of the ABA(UA) algorithm is at most $(1-\epsilon/2-d\epsilon)^j$ times its original weight.

We now show that the expected fault ratio decreases exponentially with the number of iterations. Let p_0 and q_0 be the initial weights of non-faulty and faulty processes such that $q_0/p_0 = r_0$. Let S_j and T_j be the random variables to denote weights of the non-faulty processes and faulty processes after j iterations of the ABA(UA) algorithm. Since the expected weight of each non-faulty process after j iterations is at least $(1 - \epsilon/2 + d\epsilon)^j$ times its original weight; by linearity of expectation,

$$E[S_j] \geq p_0 \cdot (1 - \epsilon/2 + d\epsilon)^j.$$

Similarly,

$$E[T_j] \leq q_0 \cdot (1 - \epsilon/2 - d\epsilon)^j.$$

We now bound $E[S_j/T_j]$. Using independence of S_j and T_j, we get that

$$E[S_j/T_j] = E[S_j] \cdot E[1/T_j].$$

We now use the fact that for any non-negative random variable X, $E[1/X] \geq 1/E[X]$ which can be shown using Jensen's inequality ($E[f(X)] \geq f(E[X])$

for convex f). Therefore,

$$\begin{aligned} E[S_j] \cdot E[1/T_j] &\geq E[S_j] \cdot 1/E[T_j] \\ &\geq \frac{p_0 \cdot (1-\epsilon/2+d\epsilon)^j}{q_0 \cdot (1-\epsilon/2-d\epsilon)^j} \\ &= 1/r_0 \cdot \left(1 + \frac{2d\epsilon}{1-\epsilon/2-d\epsilon}\right)^j \end{aligned}$$

By defining

$$k = \left(1 + \frac{2d\epsilon}{1-\epsilon/2-d\epsilon}\right),$$

we get the desired result. Because $0 < \epsilon < 1$ and $0 < d < 1/2$, $(1-\epsilon/2-d\epsilon)$ is guaranteed to be positive which ensures $k > 1$. □

Remark: The above theorem can be generalized to the case when non-faulty processes are accurate with probability at least $1/2 + d_1$ and faulty processes are accurate with probability at most $1/2 - d_2$. In this case,

$$k = \left(1 + \frac{\epsilon(d_1+d_2)}{1-\epsilon/2-d_2\epsilon}\right).$$

When $d_1 = d_2$, we get the original value of k. Also, when $d_2 = -d_1$ (faulty processes are as accurate as non-faulty processes), we get k equals 1.

3.4.3 At-Least-One Accuracy

For this section, we assume that there is at least one process in the system that is inaccurate *only* for a small number of iterations of the ABA algorithm. This assumption is sufficient to guarantee *cumulative accuracy*, i.e., a bound on the total number of mistakes made by the algorithm. Our results

are based on the method of weighted majority with multiplicative updates [31]. We first consider ABA(UI) algorithm. In the following theorem, we show that ABA(UI) guarantees accuracy for a large number of iterations.

Theorem 10. *(At-Least-One Accuracy, ABA(UI)) Assume $n \geq 3f + 1$. If there exists at least one process such that is inaccurate at most b out of j iterations of the ABA(UI) algorithm, then the algorithm is inaccurate at most $2(1 + \epsilon)b + (2/\epsilon) \log n$ times.*

Proof. The proof follows from standard arguments in multiplicative update method [31]. We initialize the weights of all the processes to $1/n$. Let $\phi(i)$ be the sum of all the weights of the processes at the end of iteration i. Suppose that for any iteration i, the ABA(UI) algorithm is wrong. This means that the weighted majority of the values in the proposed vector were wrong and hence a majority of the weights will decrease by $(1-\epsilon)$ of their previous value. Therefore,

$$\phi(i) \leq \phi(i-1)/2 + \phi(i-1)/2 \cdot (1-\epsilon) = \phi(i-1)(1-\epsilon/2).$$

The total weight, at the beginning of the algorithm, $\phi(0)$ is equal to one. Suppose that the ABA(UI) algorithm makes $m(j)$ mistakes in the first j iterations. After j iterations of ABA(UI), we get

$$\begin{aligned} \phi(j) &\leq \phi(0)(1-\epsilon/2)^{m(j)} \\ &= (1-\epsilon/2)^{m(j)}. \end{aligned}$$

Now, consider a non-faulty process that is inaccurate at most b out of j iterations. In spite of the presence of Byzantine processes, ABA(UI) algorithm guarantees that this process is never penalized when it is accurate. After j iterations, the weight of this process is at least

$$(1-\epsilon)^b \cdot (\text{its initial weight}) = (1-\epsilon)^b/n.$$

This weight is less than the total weight. Therefore,

$$(1-\epsilon)^b/n < (1-\epsilon/2)^{m(j)}.$$

Taking log on both sides and shifting n to the right hand side, we get

$$b\log(1-\epsilon) < \log n + m(j)\log(1-\epsilon/2).$$

Dividing both sides by $\log(1-\epsilon/2)$ which is a negative quantity and rearranging gives us

$$m(j) < b\log(1-\epsilon)/\log(1-\epsilon/2) - \log n/\log(1-\epsilon/2).$$

In the following part of the proof, we use two inequalities: $-\log(1-\epsilon) \leq \epsilon + \epsilon^2$ and $-\log(1-\epsilon/2) \geq \epsilon/2$ that require $\epsilon < 0.684$. Applying these inequalities we get,

$$m(j) < b \cdot 2 \cdot (\epsilon+\epsilon^2)/\epsilon + 2\log n/\epsilon.$$

Therefore,

$$m(j) < 2(1+\epsilon)b + 2/\epsilon \log n.$$

□

Interestingly, the result holds even when we use ABA(UA).

Theorem 11. *(At-Least-One Accuracy, ABA(UA)) Assume $n \geq 3f + 1$. If there exists at least one process such that is inaccurate at most b out of j iterations of the ABA(UA) algorithm, then the algorithm is inaccurate at most $2(1 + \epsilon)b + (2\epsilon) \log n$ times.*

Proof. Note that even when we update weights on all iterations, the following inequalities hold. The total weight in the system,

$$\phi(j) \leq \phi(0)(1 - \epsilon/2)^{m(j)} = (1 - \epsilon/2)^{m(j)}.$$

The weight of the process that is wrong b out of j iterations is

$$(1 - \epsilon)^b \cdot (\text{its initial weight}) = (1 - \epsilon)^b/n.$$

Hence, the previous proof applies. □

Substituting $b = 0$ in the above theorem, i.e., when at least one process is accurate for all j iterations of the algorithm, the ABA algorithm makes a mistake only $O(\log n)$ times. Note that this is independent of the number of iterations and hence, if the ABA algorithm is run for a large number of iterations ($j >> \log n$), then it guarantees accuracy in most of them. Or in other words, the ABA algorithm is approximately as accurate as the most accurate process in the system.

3.5 ABA Algorithm with Weighted Byzantine Agreement

In the ABA algorithm proposed in Figure 5, we have used standard Byzantine Agreement in Step 2. Since standard Byzantine Agreement assumes $n \geq 3f + 1$, the ABA algorithm also made the same assumption. This assumption is crucial for correctness of the ABA algorithm because agreement requires that processes have identical V vector after step 2. We now consider the case when $n < 3f + 1$, but the initial fault ratio is less than 1/2. Thus, more than a third of the processes may be faulty but the total weight of the faulty processes is still less than 1/2 of the weight of the non-faulty processes. Under this scenario, we propose an alternative to ABA algorithm by replacing Step 2 of the ABA algorithm by

▷ Step 2'(Alternative to Step 2) : Agree on V vector

for $j = 1$ to N **do**

$V[j] \leftarrow$ WeightedByzantineAgreement($V[j]$)

end for

Thus, we use the weight vector even to agree on the value of $V[j]$ (as used by the algorithms in [25]). We refer to this algorithm as ABAW algorithm. Since the fault ratio decreases under various accuracy assumptions, the ABAW algorithm works correctly even when the set of processes that act Byzantine increases with time so long as the fault ratio stays less than 1/2. The following theorem can be shown for the ABAW algorithm analogous to that for ABA algorithm.

Theorem 12. *Assuming $r_0 < 1/2$, all iterations of the ABAW algorithm guarantee agreement and termination if the weight update method ensures $r < 1/2$.*

For the deterministic accuracy property, we have the following theorem.

Theorem 13. *If the accuracy ratio $\alpha > 1/2 + d$ and if the initial fault ratio $r_0 < 2d$, for any $0 \leq d \leq 1/4$, for all iterations, then the ABAW(UI) algorithm guarantees accuracy.*

Note that Theorem 7 does not hold for ABAW(UI) because we require $r_0 < 1/2$. Theorem 8 holds for ABAW(UA) without the assumption of $n \geq 3f + 1$ (assuming $r < 1/2$). Theorem 10 does not hold for ABAW(UI) or ABAW(UA) because the fault ratio may increase beyond 1/2 even if one process is accurate most of the times.

3.6 Experimental Evaluation of ABA Algorithm

The experimental evaluation compares three different update methods: "always update", "update on inaccuracy" and "never update". The last option reduces to standard Byzantine agreement. The performance of the three accuracy models presented in this chapter are considered with each of these update methods for two different Byzantine fault models. Always update and never update perform very well under one of the fault models each, while they both perform very poorly for the other. Update on inaccuracy, the method followed in this chapter, is always close to the best.

3.6.1 Experimental Setup and Parameters

For the experimental evaluation, we focus on faulty processes that will always try to make the system agree upon an incorrect value. The faulty processes have complete knowledge of the system including the correct value for each iteration. Our simulation uses two models for faulty processes. Model 1 uses a process that will always propose the incorrect value. Model 2 uses a process that looks at the percentage of its own weight to the weight of all processes and proposes the correct value if its percentage is below a threshold and the incorrect value otherwise. There are two types of non-faulty processes used. The first is an accurate non-faulty process that always proposes the correct value ($d = 0.5$, $\beta = 1$). The second type of non-faulty process chooses the correct value with probability $\beta = 0.5 + d$, where $d \in [0, 0.5]$. The Queen algorithm [4] is used for step 2 in the ABA algorithm and for all simulations, $n = 41$, $f = 10$ and $\epsilon = 0.1$.

3.6.2 Results

Simulation results for *deterministic accuracy* are shown in Figure 3.1. For this experiment, we had one accurate process, and the other non-faulty processes had a value of $d = 0.00001$. We compare the % of accurate decisions made by the algorithm for 100 iterations, with increasing values of $a_0/(p + q)$ i.e. the starting weights of the accurate processes divided by the total weight of processes in the system. The experiments were performed for the two fault models 1 and 2. As can be seen, having an update method performs much

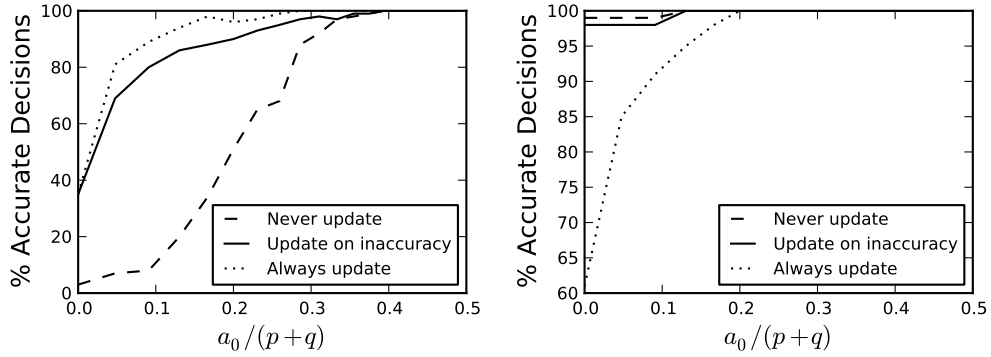
better than not having one with model 1 and always updating performs poorly with model 2. Update on error gives a good compromise between the two.

Results for *probabilistic accuracy* are shown in Figure 3.2. For this experiment, all non-faulty processes had $d = 0.02$ and all processes start with uniform weights. We compare the % of accurate decisions with increasing number of iterations. Notice that, on the whole, update on inaccuracy performs the best for these graphs. Always updating seems like the natural method to use; but, in Figure 3.2(b) always update performs the worst.

Simulation results for *at-least-one accuracy* are shown in Figure 3.3. For this experiment, we had one non-faulty process which is always accurate i.e. $d = 0.5$, and the remaining non-faulty processes had $d = 0.00001$. The processes start with uniform weights. We compare the % of accurate decisions with increasing number of iterations. For model 1 in Figure 3.3(a), updating weights increases the accuracy over iterations. With model 2 always update shows the worse performance with update on accurate being the best. Notice how update on inaccuracy is always close to the best.

3.7 Conclusion and Future Work

We introduce the problem of Accurate Byzantine Agreement with Feedback where in addition to agreeing on the same value, the processes in the system have to agree on the correct value. The notion of correctness is based on the environment or any kind of external feedback common to all the processes in the system. We present an algorithm that solves the problem for various



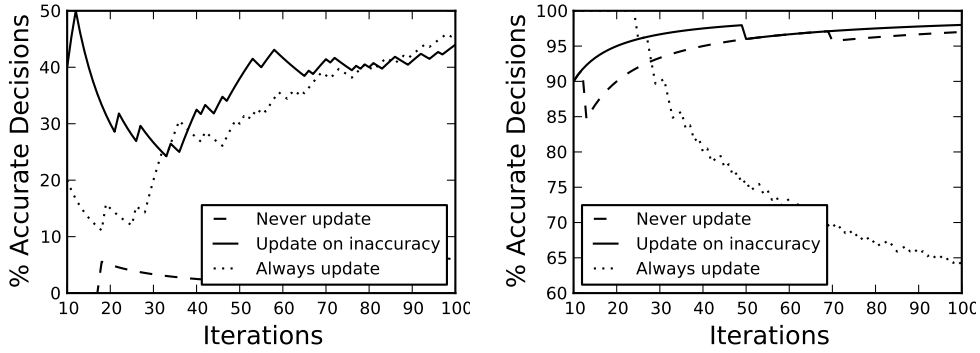
(a) With model 1.

(b) With model 2.

Figure 3.1: Deterministic accuracy: Ratio of Accurate Process Weights vs. % Accurate Decisions

assumptions on the initial accuracy and weight distribution of the processes. We show that if the weight of the accurate processes is greater than 3/4 the weight of the non-faulty processes, then the algorithm always decides on the correct value. We relax this further and show that if a majority of the non-faulty processes are accurate, then for certain assumptions on the faulty and non-faulty processes, the algorithm never makes a mistake. Further, we show that if the probability of accuracy of the non-faulty process is greater than 1/2, then the algorithm's accuracy improves exponentially in the number of mistakes it makes. Finally, we consider the simple assumption that at least one process always proposes the correct value for all iterations and show that the algorithm rarely makes mistakes.

We performed simulations comparing the performance of three different weight update methods: update on inaccuracy, always update and never update (just standard Byzantine agreement). The experiments compared the



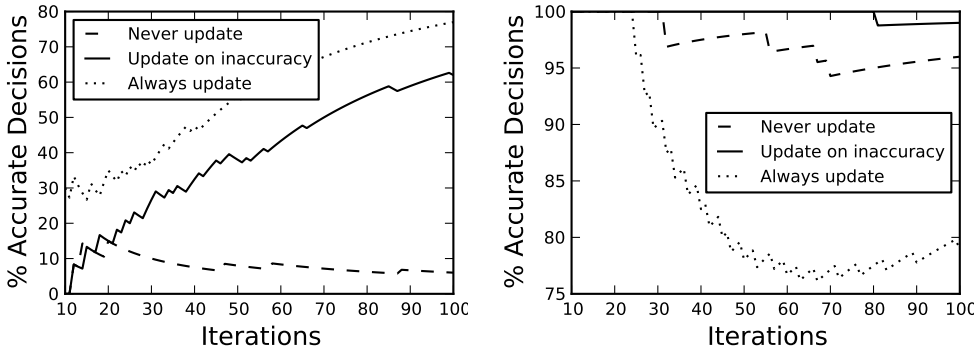
(a) With model 1.

(b) With model 2.

Figure 3.2: Probabilistic Accuracy: Iterations vs. % Accurate Decisions, $d = 0.02$

performance of these solutions under all three accuracy assumptions and the results indicate that while never update and always update perform very well for different fault models, update on inaccuracy performs uniformly well for both fault models.

This problem brings forth further questions. The results in this chapter mainly present upper bounds for the problem of accurate Byzantine agreement with feedback. We also need to explore lower bounds for the problem. Also, our results depend on the multiplicative update rule. We wish to explore other update rules, such as additive updates and compare their performance, both theoretically and practically. Designing optimal policies to guarantee maximum probability of correct decisions is also an interesting problem.



(a) With model 1.

(b) With model 2.

Figure 3.3: At-Least One Accuracy: Iterations vs. % Accurate Decisions, One accurate process

Chapter 4

Vectorized Byzantine Agreement

4.1 Introduction

The Multi-dimensional Approximate Byzantine Agreement (MDABA) [39, 47, 48] problem is approximately agreeing upon a vector where every process proposes a vector and up to t processes can be arbitrarily faulty. It is required that the final vector at all non-faulty processes be within the convex hull of the proposed vectors of non-faulty processes. Performing approximate agreement on each dimension separately does not satisfy the convex hull criteria. This can be important to some problems as satisfying convexity means the output is valid. For example, if the vectors represented empirically measured probability distributions, then satisfying the convex constraint means that the output is a convex combination of the non-faulty measurements and is a valid probability distribution. This chapter presents an equivalence between a piece of the MDABA and the center point from combinatorial geometry.

Byzantine Agreement (BA) problem was originally proposed by Lamport, Shostak and Pease [35,42]. There is extensive literature on the Byzantine Generals problem and its variations [1, 11, 13, 15, 20, 24, 32]. The basic problem is for n processes to agree upon a non-trivial output value in the presence of

t arbitrarily faulty processes. It has been proven by Fischer, Lynch, and Paterson [21] that exact agreement of this form is impossible in an asynchronous system with even one fault. Dolev, Lynch, Pinter Stark, Eugene and Weihl [14] proved that while exact agreement is impossible, approximate agreement is possible in an asynchronous system. Delev, et al. give an algorithm for approximately agreeing on a scalar value in asynchronous systems. The next step is to consider the case of a vector rather than a scalar.

A key result shown by Mendes and Herlihy [39] and Vaidya and Garg [48] using two related theorems from combinatorial geometry is that for any algorithm to guarantee that approximate agreement can hold requires $n \geq \max\{3f+1, (d+1)f+1\}$. They both show this result to guarantee that they can calculate a point inside the convex hull of the points from non-faulty processes. For example, Mendes and Herlihy [39] perform this by taking all subsets of size $n-f$ from the input points and intersecting the convex hulls of these subsets and call this the safe set. Vaidya and Tseng [47] present a more abstract version of this problem under crash faults. Their extension has the same input but outputs a convex hull at the end instead. Their version can be used to solve the Multi-dimensional Approximate Agreement problem in the presence of crash faults. The results presented here are applicable to their algorithms as well.

4.2 Definitions

The following is a set of definitions that are used in the result of this chapter.

Definition 2 (hyperplane)**.** *For any hyperplane H, a vector h can be found that fully defines H as follows:*

$$H = \forall x \in \mathbb{R}^d \| \langle h, x \rangle = 1.$$

In $\mathbb{R}^2$, a hyperplane is the line $h_1 x_1 + h_2 x_2 = 1$.

Definition 3 (half-spaces)**.** *Let H be a hyperplane of $\mathbb{R}^d$ defined by vector h. The set $H^+ = \{x : x \in \mathbb{R}^d, \langle h, x \rangle \geq 1\}$ is the positive closed half-space defined by vector h. H^{+c} is the complement of H^+ which is the negative open half-space defined by vector h.*

A positive closed half-space in $\mathbb{R}^2$ is the area including and above the line $h_1 x_1 + h_2 x_2 = 1$.

Definition 4 (Convex hull)**.** *Let P be a set of points in $\mathbb{R}^d$. The convex hull of P, denoted $Conv(P)$, is the intersection of all closed half-spaces that contain all points in P.*

The following definition is from Mendes and Herlihy [39].

Definition 5 (Restriction)**.** *Let X be a set of n points in $\mathbb{R}^d$. A* restriction *of set X is a subset $X' \in X$ containing exactly $|X| - t$ elements. The set of all possible restrictions is written $Restrict_t(X)$.*

Definition 6 (Safe area)**.** *The* safe area *of set X is defined as:*

$$Safe_t(X) = \bigcap_{X' \in Restrict_t(X)} Conv(X')$$

Definition 7 (β-center)**.** *Let P be a set of n points in $\mathbb{R}^d$. A point x, not necessarily in P, is called a* β-center *of P if all open half-space that excludes x contains less than $(1-\beta)n$ points of P.*

x a β-center if:

$$\forall H, x \in H^+ \implies |H^{+c} \cap P| < (1-\beta)n$$

A β-center where $\beta = \frac{1}{d+1}$ is special and is called a centerpoint. Helly's and Radon's theorems can be used to prove that every set of points has a $\frac{1}{d+1}$-center and that for any $\beta > \frac{1}{d+1}$ a β-center is not guaranteed to exist. For proofs of this result and further reference, see a book on combinatorial geometry like Edelsbrunner [17].

Definition 8 (β-center set)**.** *Let P be a set of n points in $\mathbb{R}^d$. Let $Cent_\beta(P)$ be the set such that for all $x \in Cent_\beta(P)$, x is a β-center of P.*

$$Cent_\beta(P) = \{x : \forall H \ s.t. \ |H^{+c} \cap P| < (1-\beta)n, x \in H^+\}$$

4.3 Results

The result of this chapter is the following theorem.

Theorem 14. *Let $n = |P|$, then*

$$Safe_t(P) = Cent_{\frac{t}{n}}(P)$$

Proof. **Case** $Safe_t(P) \subseteq Cent_{\frac{t}{n}}(P)$: Consider the set $\mathcal{H}$ of all half planes that contain all the elements of at least one set in $Restrict_t(P)$. By the definition of $\mathcal{H}$ and $Restrict_t(P)$, the intersection of any half plane in $\mathcal{H}$ with P will contain at least $n - t$ elements, as every set in $Restrict_t(P)$ has exactly $n - t$ elements. In other words, $\mathcal{H}$ is the set of all half-planes that satisfy:

$$\forall H^+ \in \mathcal{H}, |H^+ \cap P| \geq n - t.$$

By contrapositive, the complement of every half plane in $\mathcal{H}$ intersected with P will contain less than $n - t$ elements of P. In other words, $\mathcal{H}$ is the set of all half-planes that satisfy:

$$\forall H^+ \in \mathcal{H}, |H^{+c} \cap P| < n - t.$$

$Safe_t(P)$ is the intersection of convex hulls. Any half plane that contains one of the convex hulls in the intersection will contain the intersection. Therefore, every half plane in $\mathcal{H}$ contains all points in $Safe_t(P)$. The intersection of all half-planes in $\mathcal{H}$ can be written as:

$$\{x : \forall H \text{ s.t. } |H^{+c} \cap P| < n - t, x \in H^+\}.$$

This is $Cent_{\frac{t}{n}}(P)$. Therefore, $Safe_t(P) \subseteq Cent_{\frac{t}{n}}$.

Case $Cent_{\frac{t}{n}} \subseteq Safe_t(P)$: A convex hull can be completely described by the intersection of all half-planes that contain it. The intersection of two convex hulls then is the intersection of the half-planes that describe both convex hulls. Let $\mathcal{H}$ be the set of all half-planes that contain a set in $Restrict_t(P)$. $Safe_t(P)$ is the intersection of all half-planes in $\mathcal{H}$. Because every half plane in $\mathcal{H}$ contains a set in $Restrict_t(P)$, every half-plane contains at least $n - t$ points of P. $Cent_{\frac{t}{n}}$ is described by the intersection of all half-planes that contain at least $n - t$ points of P. Therefore, $Cent_{\frac{t}{n}} \subseteq Safe_t(P)$.

□

Now notice that any algorithm from combinatorial geometry that can be used to compute a β-center set can be used to compute the safe area for multi-dimensional approximate agreement. For example, a β-center approximation algorithm in polynomial time in d is given by Clarkson, Eppstein, Miller, Sturtivan, and Teng [10]. Their β-center approximation algorithm computes $\beta = \frac{1}{d^2}$-centers in polynomial time. Both [39, 48] give the bound for a solution to be guaranteed to be $n \geq \max 3f + 1, f(d + 2) + 1$. This bound changes to $n \geq \max 3f + 1, f(d^2 + 2) + 1$ if that approximation algorithm is used.

Also, the following observation gives additional insight into the safe areas structure.

Lemma 12. *Projecting the set $Cent_{\frac{t}{n}}(P)$ onto an arbitrary line l is the same*

interval as removing the t smallest and t largest values of P projected onto that line.

Proof. A direct result of the definition of the β-center set is that there will exist two half-spaces that contain $Cent_{\frac{t}{n}}(P)$ and are normal to l that both have exactly $n - t$ points of P. □

This result implies that doing multi-dimensional approximate byzantine agreement has a similar structure as doing the one dimensional problem, but, in every possible direction.

4.4 Conclusion

This chapter has proven that the safe area needed for Multi-Dimensional Approximate Byzantine Agreement is the same β-center. This observation can be used adapt centerpoint algorithms from combinatorial geometry to solve for a point in the safe area more effectively. Also, there exist approximate centerpoint algorithms that can be applied to MDABA.

Chapter 5

Error Correction Codes in Repeated Broadcast Communication

5.1 Introduction

Many distributed algorithms require a step in which every participating process needs a value from every other process. For example, in a clock synchronization algorithm, every process may collect the values of clocks of all other processes. In a sensor network, a group of sensors may collect values from each other to compute the average value, or some other global function such as the minimum, the maximum, or the sum of all the values. In a system that requires a uniform action, the processes may collect proposals from all other processes to determine an action. This chapter addresses these problems in the presence of Byzantine failures. Many fault tolerant algorithms need to have information about what other processes know about other processes. This is call second-order information. In order to perform a fault tolerant broadcast, second-order knowledge is required. The usual method to acquire second-order information is for every process to broadcast the informa-

The work presented in this chapter is based on the following publication.
John F. Bridgman and Vijay K. Garg. All-to-all gradecast using coding with byzantine failures. Technical Report TR-PDS-2012-001, Parallel and Distributed Systems Laboratory, The University of Texas at Austin, 2012.

tion that they have; then, every process rebroadcasts what they receive. But, rebroadcasting the information is inefficient when it is known that the number of faulty processes is bounded. The technique described in this chapter uses a forward error correction (FEC) code to minimize the size of the messages that are rebroadcast. As an example, an application of this technique to gradecast is given. [7, 8] Gradecast can be used as a basic building block for many distributed algorithms that handle Byzantine failures.

The gradecast algorithm, first proposed by Feldman and Micali [19], is a broadcast algorithm that gives the receivers a confidence level in the value received. Let $value_j[k]$ be the value that process P_j outputs for process P_k, $confidence_j[k]$ be the confidence value process P_j outputs for process P_k, and v_k be the initial input value to the algorithm for process P_k. The confidence level returned is from the set $\{0, 1, 2\}$ and the confidence value gives information about the state of the other processes. The gradecast algorithm provides three main properties of the confidence level that allow a process to reason about the knowledge of other processes.

1. For all non-faulty process P_i, and non-faulty process P_j, and any process P_k, if $confidence_j[k] > 0$ and $confidence_i[k] > 0$; then, $value_j[k] = value_i[k]$.

2. For any non-faulty process P_i, and non-faulty process P_j, and any process P_k, $|confidence_i[k] - confidence_j[k]| \leq 1$.

3. If P_k is non-faulty, then for all non-faulty processes P_i, $confidence_i[k] = 2$ and $value_i[k] = v_k$.

The original one-to-all gradecast algorithm broadcasts a value from one process to all the other processes. Message bit complexity is defined as the total number of bits sent by all non-faulty processes in one invocation of the algorithm. The one-to-all gradecast algorithm has a message bit complexity of $O(mn^2)$, where m is the length of the message and n is the number of processes. The properties of gradecast make it a useful primitive in distributed systems.

Consider the case where all processes wish to broadcast a value to all other processes using gradecast. This is referred to as all-to-all gradecast and it is used in many applications such as Byzantine agreement, approximate agreement, and multiconsensus [3]. The standard implementation of all-to-all gradecast, where n instances of the one-to-all gradecast algorithm are used, has $O(mn^3)$ message bit complexity. This chapter shows a method, using coding, that gives an all-to-all gradecast algorithm with only $O(mtn^2)$ message bit complexity, where t is the specified maximum number of faulty processes. This is a significant reduction in message bit complexity when t is much smaller than n, which is usually the case. Furthermore, gradecast requires $t < n/3$ for correctness.

The all-to-all gradecast algorithm in this chapter uses error correction codes [45] to mask Byzantine failures and has wide applicability in distributed

systems. For example, by replacing the original gradecast in the byzantine agreement algorithm proposed by Ben-Or, Dolev and Hoch [3] with $O(mtn^3)$ message bit complexity, a new byzantine agreement algorithm with $O(mt^2n^2)$ message bit complexity results. If the number of actual failures is $f \leq t$, then, the algorithm by Ben-Or, Dolev and Hoch will take $\min(f+2, t+1)$ rounds. This property is often referred to as early stopping. The bit complexity of approximate agreement algorithm [3, 14, 18] is reduced from $O(kn^3)$ to $O(kn^2t)$, where k is the number of rounds used in the approximate algorithm. Algorithms that have better message bit complexity exist; but, they sacrifice round complexity or reduce the maximum number of faulty processes tolerated. The example byzantine agreement algorithms in this chapter are given because of the simplicity of their implementation on top of an all-to-all gradecast algorithm. There exist algorithms with better message bit complexity. For example, the algorithm by Coan and Welch [12] has message bit complexity of $O(t^2 + nt)$ to agree on a single bit. This algorithm does not posses an early stopping property.

Error correction codes can be viewed as a projection from a smaller space to a larger space with good separation. Because the points in the larger space are separated, small perturbations in the point in the larger space are still close to the original mapped point and the point in the original smaller space can be recovered. Generally, the spaces are high dimensional vector spaces over finite fields and the measure of distance between two elements of the space is the number of coordinates that have a different value. *Systematic* codes can be

constructed that encode a vector as the original vector concatenated with an error correction vector. The method presented here relies on the observation that every process is sending a value to every other process and only the faulty processes will send conflicting data. So, the vector built at each process will differ in at most t locations. This can be viewed as transmitting the vector and each process receiving a corrupted version. Then, only the error correction part of the encoded vector need be sent between processes to correct these "errors". The original vector is not actually transmitted. In a traditional application of error correction codes, an input block is encoded and then the whole output codeword is transmitted. The whole codeword is not transmitted. Only a portion of the codeword is transmitted. A proper selection of the code allows an error correction vector that can correct t errors to be no longer than $2t+1$.

This method of using coding is also applicable to other types of broadcast algorithms. Srikanth and Toueg [46] give a broadcast algorithm to simulate authenticated broadcasts that has the important properties of authenticated messages. These are as follows: If a correct process P_i broadcasts a message; then, all other correct processes receive that message and if a correct process P_i does not broadcast a message; then, no correct process receives a message from P_i. The message bit complexity of a consistent broadcast is $O(mn^2)$ and therefore, the message bit complexity of all-to-all consistent broadcast is $O(mn^3)$. By using the method here, the bit complexity of all-to-all consistent broadcast can be reduced to $O(mtn^2)$.

All-to-all gradecast can also be used to implement an interactive con-

sistency algorithm. Interactive consistency [29, 42] is the problem in which each process has a vector with an entry that needs to be filled from every other process and all vectors should be the same at the end of the algorithm. Interactive consistency is at least as difficult as Byzantine agreement.

There are earlier works that use error correcting codes for Byzantine broadcast algorithms. Liang and Vaidya [37] give an algorithm that achieves communication complexity of $O(mn)$ bits for broadcast with Byzantine failures if $m = \Omega(n^6)$. This is quite useful in situations where the message being broadcast is a very long stream of bits. An example of such messages is all the samples from a sensor in a long running system. However, for small message size, m, the communication complexity is $O(nm + n^4 m^{1/2} + n^6)$. This work is most useful when every process is doing a broadcast and the message size may not be large. Friedman, Mostéfaoui, Rajsbaum and Raynal [23] show a mapping from a distributed agreement problem to a coding problem. This approach is to use coding to reduce the size of the messages being sent. The work by Krol [33] gives a set of algorithms that use coding to perform Byzantine consensus. Essentially, Krol [33] replaces broadcast with encoding, and decision making with decoding. These algorithms are based on the original algorithm by Pease, Shostak and Lamport [35] and have exponential message complexity.

The remainder of this chapter is organized in the following manner. First, an overview of the original algorithm is given in section 5.2. The algorithm is described in section 5.3. Next, proofs of its correctness are in section

5.5. Then, in section 5.6, applications of an all-to-all gradecast algorithm are discussed. Concluding remarks are in section 5.7.

Table 5.1: Notation in Chapter 5

n	number of processes
t	maximum number of faulty processes
i, j, k	process IDs
u, v, w, x, y, z	scalar values
U, V, W, X, Y, Z	non-scalar values
$confidence_i[j]$	confidence value P_i has in P_j's value
$value_i[j]$	value process P_i received from process P_j
G	set of all non-faulty processes

5.2 One-to-All Gradecast

This section gives a quick overview of the original algorithm presented by Feldman and Micali [19]. This algorithm broadcasts a value from one process to all other processes. This can be modified to an all-to-all gradecast algorithm by vectorizing. Pseudo-code for the algorithm is in Figure 6. It assumes that the values n, t, and h are common knowledge to all processes, where n is the number of processes, t is the maximum number of faulty processes, and h is the broadcasting process. The algorithm proceeds in four steps. In the first step, the broadcaster h sends out its value to all processes. After this step, the algorithm is symmetric. In the second step, all process rebroadcast the value received from h to all other processes. Then, in the third step, each process looks at the values received from Step 2. If there is a common value that has been received at least $n - t$ times; then, it broadcasts that value.

Otherwise, the process broadcasts no value. Finally, in Step 4, the received values from Step 3 are examined. Let x be the value that appears the most in Z_i. If there is a tie between two values, some common agreed upon tie breaking strategy must be performed. For example, if values are real numbers, one can always take the minimum. If x appears at least $2t+1$ times; then, P_i outputs x with confidence 2. If x appears less than $2t+1$ and more than t times; then, P_i outputs x with confidence 1. Otherwise, P_i outputs $\perp$ with confidence 0.

This algorithm has message bit complexity $O(mn^2)$ and when replicated to perform all-to-all gradecast, will have $O(mn^3)$ message bit complexity. The next section a vectorized modification to this algorithm is shown that reduces the all-to-all gradecast message bit complexity to $O(mtn^2)$.

5.3 Algorithm for All-To-All Gradecast

This section gives the all-to-all gradecast algorithm that has $O(mtn^2)$ message bit complexity. This algorithm is based on vectorizing the gradecast algorithm presented by Feldman and Micali [19]. As before, each process P_i has an input value v_i and the algorithm produces two vectors $value_i$ and $confidence_i$ which are the received values and the confidence level respectively. The algorithm assumes that the set of all messages can be encoded as members of a finite field, with one field member reserved to represent "no message" which will be denoted as $\perp$. This assumption only requires that there exists a mapping between the messages and the field elements such that every message has a unique field element assigned to it with at least one field element

Algorithm 6 Original One-to-all Gradecast Algorithm

▷ Inputs to P_h

1: $v_h \leftarrow$ input value for broadcaster

▷ Variables

2: $u_i \leftarrow$ value process P_i receives in Step 2
3: $X_i[1..n] \leftarrow$ vector of values received in Step 3
4: $Z_i[1..n] \leftarrow$ vector of values received in Step 4

▷ Step 1

5: **if** $i = h$ **then**
6: BROADCAST(v_h)
7: **end if**

▷ Step 2

8: $u_i \leftarrow$ RECEIVE(P_h)
9: BROADCAST(u_i)

▷ Step 3

10: **for** $j = 1$ **to** n **do**
11: $X_i[j] \leftarrow$ RECEIVE(P_j)
12: **end for**
13: **if** $\exists x$ **such that** $|\{k : X_i[k] = x\}| \geq n - t$ **then**
14: BROADCAST(x)
15: **end if**

▷ Step 4

16: **for** $j = 1$ **to** n **do**
17: **if** P_j sent a message to P_i **then**
18: $Z_i[j] \leftarrow$ RECEIVE(P_j)
19: **else**
20: $Z_i[j] \leftarrow \perp$
21: **end if**
22: **end for**
23: **if** $\max_x |\{k : Z_i[k] = x\}| \geq 2t + 1$ **then**
24: **return** $\arg\max_x |\{k : Z_i[k] = x\}|$ with confidence 2
25: **else if** $\max_x |\{k : Z_i[k] = x\}| > t$ **then**
26: **return** $\arg\max_x |\{k : Z_i[k] = x\}|$ with confidence 1
27: **else**
28: **return** $\perp$ with confidence 0
29: **end if**

unassigned.

There is a standard technique, called interleaving, to apply a small code to larger blocks without increasing the code length. The tool Parity Archive Volume Set [40] uses this technique. The usage of this technique relies on the fact that only t blocks may be corrupt. It is very similar to breaking up the message to be transmitted into blocks and running each block through the code, except it is broken into interleaved blocks. What this means for the problem here, is that, if a code that uses octets as the basic unit and one message is ten octets; then, the first block will be the first octet from each message in the vector of messages, the second block will be the second octet from each message, and so on. Note, for the purposes here, the blocks are only interleaved in this manner for the encoding and decoding process. For example, if the vector to encode is $[[a, b], [c, d], [e, f]]$; then, $[a, c, e]$ would be run through the encoder to produce $[a, c, e, g, h]$ and $[b, d, f]$ to produce $[b, d, f, i, j]$ and the final output of the encoder is then $[[a, b], [c, d], [e, f], [g, i], [h, j]]$, which is then used in the algorithm. With this method, messages longer than the field size can be used.

Pseudo-code for the algorithm is provided in Figure 7 and Figure 8. This algorithm proceeds in four steps. The following description is from the point of view of process P_i, because the algorithm is symmetric. First, in Step 1, P_i broadcasts its value to every other process. Step 2 starts to differ from the original gradecast algorithm. The original algorithm rebroadcasts the values received from Step 1. Because of the messaging system reliability, the property

$\forall P_i, P_j, P_k \in G : V_i[k] = V_j[k]$, where G is the set of all non-faulty processes is guaranteed. This implies that $\forall P_i, P_j \in G : |\{k : V_i[k] \neq V_j[k]\}| \leq t$. This means that at least $n - t$ values between non-faulty processes are identical; so, sending the whole vector, V_i, is inefficient. Therefore, the algorithm here uses coding techniques to send at most $2t + 1$ values, which can be used in conjunction with the knowledge that the receiving process possesses to recover everything the sender knows. To finish Step 2, P_i sends the error correction vector of V_i.

In Step 3, P_i receives the encoded message from all other processes and uses its current knowledge to construct matrix X_i of all the values that every process claims that every other process possesses as their input value. The value $X_i[j][k]$ is the value that j claims k sent to it. The reliability of the messaging system and how the coding process works implies that $\forall P_i, P_j \in G, \forall k : X_i[j][k] = V_j[k]$. Now an array Y_i is constructed from X_i in the following manner. For each P_j, if there is a value that appears at least $n-t$ times in the column $X_i[\cdot][j]$; then, set $Y_i[j]$ to that value, otherwise, set $Y_i[j]$ to $\perp$. Then, an encoding of Y_i is sent to all processes.

Finally, in Step 4, Z_i is constructed in the same manner as X_i in Step 3. P_i uses its knowledge of Y_i and the encoded value sent to it from each other process j to recover Y_j and then places that value in the row $Z_i[j][\cdot]$. That gives the property $\forall P_i, P_j \in G, \forall k : Z_i[j][k] = Y_j[k]$. Then, P_i looks at columns of $Z_i[\cdot][j]$ for each P_j to decide its output. If $\max_x |\{k : Z_i[k][j] = x\}| \geq 2t + 1$; then, P_i sets $value_i[j] = x$ and $confidence_i[j] = 2$. If $2t + 1 > \max_x |\{k :$

$Z_i[k][j] = x\}| > t$; then, P_i sets $value_i[j] = x$ and $confidence_i[j] = 1$. Otherwise, P_i sets $value_i[j] = \perp$ and $confidence_i[j] = 0$. Notice that the reduction in message bit complexity comes from taking advantage of the knowledge that is known to be common across processes, because of the constraint that at most t processes can be faulty. The processes also do not know which of the t values are not common. This is why they must exchange information in Step 2 and 3. But, coding is used to ensure that the amount of information exchanged is small.

Algorithm 7 All-to-all Gradecast Algorithm Declarations

▷ Inputs

1: $v_i \leftarrow$ Input value for P_i

▷ Variables

2: $V_i[1..n] \leftarrow$Vector received in Step 2, initially $\perp$
3: $Vecc_i[1..2t+1] \leftarrow$error correction vector for V_i
4: $X_i[1..n][1..n] \leftarrow$Matrix of decoded values in Step 3
5: $Y_i[1..n] \leftarrow$Vector of values computed in Step 3, initially $\perp$
6: $Yecc_i[1..2t+1] \leftarrow$Error correction vector for Y_i
7: $Z_i[1..n][1..n] \leftarrow$Matrix of decoded values in Step 4
8: $value_i[1..n] \leftarrow$Vector of output values, initially $\perp$
9: $confidence_i[1..n] \leftarrow$Vector of confidence levels, initially 0

5.4 Example

The following example shows how the algorithm works. For this example, $n = 4$ and $t = 1$. The possible messages are the non-zero values over the finite field $GF(2^8)$ and the zero value is reserved to represent no message. Let P_4 be the faulty process and let the initial value for the non-faulty processes be $\{241, 86, 35\}$. For the encoder, this example will use a Reed Solomon [44]

Algorithm 8 All-to-all Gradecast Algorithm

▷ Step 1

10: BROADCAST(v_i)

▷ Step 2

11: $V_i \leftarrow$ RECEIVE-FROM-ALL
12: $Vecc_i \leftarrow$ ENCODE(V_i)
13: BROADCAST($Vecc_i$)

▷ Step 3

14: **for** $j = 1$ **to** n **do**
15: $\quad Vecc_j \leftarrow$ RECEIVE(P_j)
16: $\quad X_i[j] \leftarrow$ DECODE(V_i, $Vecc_j$)
17: **end for**
18: **for** $j = 1$ **to** n **do**
19: $\quad$ **if** $\exists x$ **such that** $|\{k : X_i[k][j] = x\}| \geq n - t$ **then**
20: $\quad\quad Y_i[j] \leftarrow x$
21: $\quad$ **end if**
22: **end for**
23: $Yecc_i \leftarrow$ ENCODE(Y_i)
24: BROADCAST($Yecc_i$)

▷ Step 4

25: **for** $j = 1$ **to** n **do**
26: $\quad Yecc_j \leftarrow$ RECEIVE(P_j)
27: $\quad Z_i[j] \leftarrow$ DECODE(Y_i, $Yecc_j$)
28: **end for**
29: **for** $j = 1$ **to** n **do**
30: $\quad count \leftarrow \max_x |\{k : Z_i[k][j] = x\}|$
31: $\quad value \leftarrow \arg\max_x |\{k : Z_i[k][j] = x\}|$
32: $\quad$ **if** $count \geq 2t + 1$ **then**
33: $\quad\quad value_i[j], confidence_i[j] \leftarrow value, 2$
34: $\quad$ **else if** $count > t$ **then**
35: $\quad\quad value_i[j], confidence_i[j] \leftarrow value, 1$
36: $\quad$ **end if**
37: **end for**
38: **return** $value_i$ and $confidence_i$

code with a code length of 2^8 that can correct one error. The error correction terms are calculated by taking the remainder of the values to encode as a polynomial with the generator polynomial $102 + 164x + x^2$ over the finite field $GF(2^8)$. For example, $[241, 86, 35, 35]$ is encoded as the polynomial $35x^{251} + 35x^{252} + 86x^{253} + 241x^{254}$. The remainder is taken, which gives us the polynomial $78+39x$, which corresponds to the values $[39, 78]$. Note that all the arithmetic operations are done over the finite field $GF(2^8)$. Decoding is much more involved and it is recommend for the reader to consult the literature on the subject [44,45]. The Schifra [41] library was used to compute these values.

For Step 1, all processes send their values to all other processes. For this example, the received values for each process are:

$$\begin{array}{rl} V_1 = & [241, 86, 35, 35] \\ V_2 = & [241, 86, 35, 35] \\ V_3 = & [241, 86, 35, 40] \end{array} \tag{5.1}$$

Encoding these gives:

$$\begin{array}{rl} [V_1,\, Vecc_1] = & [241, 86, 35, 35, 39, 78] \\ [V_2,\, Vecc_2] = & [241, 86, 35, 35, 39, 78] \\ [V_3,\, Vecc_3] = & [241, 86, 35, 40, 82, 30] \end{array} \tag{5.2}$$

Then, each process sends the $Vecc_i$ values which are of length two.

Next, for Step 3, all processes receive the values sent in Step 2. Since P_4 is faulty, it will send $[22, 77]$ to P_1, $[0, 136]$ to P_2 and $[121, 159]$ to P_3. For this example, the processes then receive:

$$\begin{array}{rl} P_1.receive(P_1) = & [39, 78] \\ P_1.receive(P_2) = & [39, 78] \\ P_1.receive(P_3) = & [82, 30] \\ P_1.receive(P_4) = & [22, 77] \end{array} \tag{5.3}$$

$$\begin{aligned} P_2.receive(P_1) &= [39, 78] \\ P_2.receive(P_2) &= [39, 78] \\ P_2.receive(P_3) &= [82, 30] \\ P_2.receive(P_4) &= [0, 136] \end{aligned} \tag{5.4}$$

$$\begin{aligned} P_3.receive(P_1) &= [39, 78] \\ P_3.receive(P_2) &= [39, 78] \\ P_3.receive(P_3) &= [82, 30] \\ P_3.receive(P_4) &= [121, 159] \end{aligned} \tag{5.5}$$

Each process concatenates the received value to the end of its V_i vector and runs this through the decoder to get:

$$X_1 = \begin{bmatrix} 241, 86, 35, 35 \\ 241, 86, 35, 35 \\ 241, 86, 35, 40 \\ 241, 49, 35, 35 \end{bmatrix} \tag{5.6}$$

$$X_2 = \begin{bmatrix} 241, 86, 35, 35 \\ 241, 86, 35, 35 \\ 241, 86, 35, 40 \\ 241, 86, 129, 35 \end{bmatrix} \tag{5.7}$$

$$X_3 = \begin{bmatrix} 241, 86, 35, 35 \\ 241, 86, 35, 35 \\ 241, 86, 35, 40 \\ 157, 86, 35, 40 \end{bmatrix} \tag{5.8}$$

Following the instructions for building Y_i in Step 3 gives:

$$\begin{aligned} Y_1 &= [241, 86, 35, 35] \\ Y_2 &= [241, 86, 35, 35] \\ Y_3 &= [241, 86, 35, 0] \end{aligned} \tag{5.9}$$

Then building $Yecc_i$ gets:

$$\begin{aligned} [Y_1,\, Yecc_1] &= [241, 86, 35, 35, 39, 78] \\ [Y_2,\, Yecc_2] &= [241, 86, 35, 35, 39, 78] \\ [Y_3,\, Yecc_3] &= [241, 86, 35, 0, 8, 182] \end{aligned} \tag{5.10}$$

Each process i then sends its $Yecc_i$. Let P_4 send $[87, 77]$ to process 1 and 2 and $[123, 149]$ to process 3.

Finally, in Step 4, each process constructs the Z_i matrix in the same way it constructed the X_i matrix. Which gives the result:

$$Z_1 = Z_2 = \begin{bmatrix} 241, 86, 35, 35 \\ 241, 86, 35, 35 \\ 241, 86, 35, 0 \\ 241, 86, 0, 35 \end{bmatrix} \tag{5.11}$$

$$Z_3 = \begin{bmatrix} 241, 86, 35, 35 \\ 241, 86, 35, 35 \\ 241, 86, 35, 0 \\ 241, 86, 35, 82 \end{bmatrix} \tag{5.12}$$

Finally, the algorithm will output for each process:

$$\begin{array}{ll} value_1 = & [241, 86, 35, 35] \\ confidence_1 = & [2, 2, 2, 2] \\ value_2 = & [241, 86, 35, 35] \\ confidence_2 = & [2, 2, 2, 2] \\ value_3 = & [241, 86, 35, 35] \\ confidence_3 = & [2, 2, 2, 1] \end{array} \tag{5.13}$$

5.5 Proof of Correctness

In this section, the correctness of the algorithm in this chapter is shown. The first lemma shows a crucial property of Y in Step 3 of Figure 8.

Lemma 13. *Assume P_i and P_j are non-faulty processes. In Step 3 of Figure 8, if P_i sets $Y_i[k]$ to $x \neq \perp$ and P_j sets $Y_j[k]$ to $y \neq \perp$; then, $x = y$. Formally,* $\forall P_i, P_j \in G, \forall k : Y_i[k] \neq \perp \wedge Y_j[k] \neq \perp \implies Y_i[k] = Y_j[k]$.

Proof. If P_i sets $x \neq \perp$ to $Y_i[k]$, then the kth column of X_i contained at least $n - t$ copies of x. Only t rows can correspond to faulty processes, so at least $n - 2t$ of the rows that contain x in column k come from non-faulty processes. This means that those $n-2t$ non-faulty processes also sent vectors to P_j which set x to the kth column for those processes. Suppose $y \neq x$ and $y \neq \perp$. This means that there must be $n - 2t$ values which are $\perp$ in the kth column of X_j at process P_j. But, $n - 2t > t$ so P_j will set $Y_j[k]$ to $\perp$, which contradicts that $y \neq \perp$ and $y \neq x$. □

Theorem 15 (Property (1))**.** *All non-faulty processes with positive confidence about process k have identical value*$[k]$*. Formally,*

$$\forall P_i, P_j \in G, \forall k : \mathit{confidence}_i[k] > 0 \wedge \mathit{confidence}_j[k] > 0$$

implies

$$\mathit{value}_i[k] = \mathit{value}_j[k].$$

Proof. First note that the Z_i matrix will contain the Y_j vectors from Step 3 of Figure 8 for all P_j. By Lemma 13, if there is a majority of a value that is not $\perp$ in the kth column of Z_i; then, all values in that column that are not the majority and not $\perp$ are from a faulty process. This implies that if any non-faulty process P_j sets $\mathit{confidence}_i[k] \geq 1$; then, all other non-faulty processes P_j that set $\mathit{confidence}_j[k] \geq 1$ also set $\mathit{value}_j[k] = \mathit{value}_i[k]$. □

Theorem 16 (Property (2))**.** *For any two non-faulty processes, the difference in their confidence levels for any process* P_k *can differ by at most* 1*. Formally,* $\forall P_i, P_j \in G, \forall k : \left|\mathit{confidence}_i[k] - \mathit{confidence}_j[k]\right| \leq 1$.

Proof. Assume some non-faulty process P_i sets $confidence_i[k] = \delta$ and $value_i[k] = x$. Process P_i setting $confidence_i[k] = \delta$ implies that a set R of processes sent x to P_i in Step 3 of Figure 8. Let $R_e \subseteq R$ be the faulty processes that sent x to P_i. By problem setup, $|R_e| \leq t$. This means that the number of processes that also sent x to any other process can differ by at most t. Let P_j be the process that receive the most messages in support of x. Then, all other processes receive at least $|R| - t$ messages in support of x. Step 4 of the algorithm in Figure 8 compares the support of x to $2t + 1$ and t to select the confidence level. By the above reasoning, the support of x differs by at most t between any non-faulty process. Therefore, the difference in confidence level between any non-faulty processes is at most 1. □

Theorem 17 (Property (3))**.** *If P_k is non-faulty, then, all non-faulty processes P_i have the value sent by process P_k and their confidence level on this value is* 2*. Formally,*

$$\forall P_i, P_k \in G : (confidence_i[k] = 2) \wedge (value_i[k] = v_k)\,.$$

Proof. If P_i is a non-faulty process, then, all processes will receive v_i from P_i in Step 2 of Figure 8. Next, all non-faulty processes will also claim that P_i sent v_i for Step 3. Let G be the set of all non-faulty processes, by the assumptions of the problem $|G| \geq n - t$ and all non-faulty processes will distribute error correction vectors with v_i in the ith entry in Step 3. So, every non-faulty process P_j will set $confidence_j[i] = 2$ and $value_j[i] = v_i$ in Step 4. □

Theorem 18. *The algorithms in Figure 8 has bit message complexity of* $O(mtn^2)$.

Proof. In Step 1, every process sends its value to every other process taking mn^2 message bits. In Step 2, each process computes $Vecc_i$ which contains at most $2t + 1$ values of length m bits. Every process then sends its $Vecc_i$ to every other process resulting in at most $m(2t + 1)n^2$ message bits. In Step 3, the same number of message bits are sent as Step 2. This results in a total of at most $mn^2 + 2m(2t + 1)n^2$ message bits being sent by this algorithm. This is $O(mtn^2)$. □

5.6 Application

The all-to-all gradecast algorithm can be used to create an exceptionally simple byzantine agreement algorithm. Ben-Or, Dolev and Hoch [3] give a simple algorithm for Byzantine agreement and approximate agreement based on the gradecast algorithm. A modification to the gradecast algorithm is needed for their Byzantine agreement algorithm. The modification is to make the algorithm take a set of known faulty processes that the algorithm will ignore and set all values for processes in the faulty set to $\perp$. This has the effect of making that process in the faulty set disappear as if they had crashed. The Byzantine consensus algorithm is symmetric, can agree upon an arbitrary value (as long as there is some method of resolving a tie), and has an early stopping property. They define early stopping to mean if there are $f \leq t$ actual failures;

then, the algorithm terminates in $\min(f+2, t+1)$ rounds. The message bit complexity with the all-to-all gradecast algorithm is $O(mt^2n^2)$.

The algorithm starts off with a faulty set which is initially empty. Then, for each round r up to $t+1$ rounds the algorithm performs as follows: The algorithm performs an all-to-all gradecast of the current value ignoring all processes in the faulty set. The algorithm then adds up how often each value was received which had a confidence greater than or equal to one. Next, it sets the current value to the value that has the largest count. If there is more than one with the same count; then, use some tie breaking scheme, such as always choosing the smaller value. The algorithm adds all processes that have confidence one or less to the faulty set. Next, the algorithm counts the number of processes that sent the current value with confidence 2. If this count is greater than $n-t$, the algorithm performs one more iteration of the loop and then exits the loop prematurely. To finish, the algorithm returns the current value.

The approximate agreement algorithm presented by Ben-Or, Dolev and Hoch is very similar to the byzantine agreement algorithm described above. The all-to-all gradecast algorithm described here can be plugged into their algorithm without changing any of the properties of the original algorithm.

5.7 Conclusion

Many algorithms have a step where every process broadcasts a value. Gradecast is a broadcast algorithm that gives a confidence level to each receiv-

ing process. This confidence level gives information about the state of other processes. An all-to-all gradecast with message bit complexity $O(mtn^2)$ is presented. The original gradecast algorithm presented by Feldman and Micali [19] is a one to all broadcast protocol. Using the original gradecast algorithm to produce all-to-all gives $O(mn^3)$ message bit complexity. This algorithm can be used in place of the original gradecast algorithm when an all-to-all broadcasts is used. The algorithm presented uses coding to reduce the amount of redundant information being transmitted. Proofs that the modified algorithm maintains the important properties of the original gradecast are given. Having an all-to-all gradecast algorithm that is efficient in message bit complexity admits a simple symmetric arbitrary valued Byzantine agreement with early stopping property that only takes $O(mt^2n^2)$ message bit complexity. Other algorithms may also benefit from using coding in the fashion presented here.

Chapter 6

Error Correction Codes for Data Structure Synchronization

6.1 Introduction

In any sort of distributed or concurrent application, synchronization between actions is imperative to correct results. This synchronization can come in many different forms ranging from atomic memory operations to locking of data structures. Every concurrent algorithm has certain requirements on order. There are some algorithms that only require basic atomic memory operations and synchronize by letting processes step on each other and duplicate work. For example, I high dimensional convex optimization problem could have multiple threads each computing random partial differentials and then adding their result to the current vector. Other algorithms create a total order of actions on a given memory region, forcing some threads to wait to enforce this total order. For example, using a mutex to protect a data structure from concurrent access. In the middle of this is the class of algorithms that are called lock free. They synchronize by using atomic memory operations with a failure path to retry the operation. This chapter presents a method of synchronization using error correction codes to allow processes to recover from processes stepping on each other.

In this chapter, a method using coding to construct a block of memory that appears to be atomically updated to the readers of the block. The basic unit is a buffer that can be read and updated atomically. Combinations of these buffers can then be used to implement more advanced algorithms. An equivalent locking algorithm is a buffer where all access is guarded by a mutex. A read-write lock that allows concurrent readers gives a small performance improvement in some cases. A simple lock-free implementation of such a buffer is copy-on-write. The writer copies the contents of the buffer into a new buffer, makes the changes; then, does an atomic compare and swap to commit the changes. These are the common ways of solving this problem.

This chapter presents an alternative approach where the buffer is simply read by the readers and the writer just writes directly into the buffer. An error correction code checksum of the buffer is kept. The readers then use this checksum to recover a consistent view of the buffer, regardless of the modifications the writer has made concurrently. Only when the writer is done, does the writer write the checksum. Only single writer and multiple readers will be considered. This is sometimes called single writer concurrency. Single writer concurrency has been considered in the literature [2].

6.2 Approach

The coded buffer gets atomic write for one thread, and atomic read for any number of threads. A simple explanation of how this works is as follows. Consider a code word as a point in a high dimensional space [44, 45]. The

property that makes an error correction code able to correct errors is that the points in this high dimensional space are far apart. Modifying the code, moves the point from the original spot towards other codes. There is always a closest code word. The maximum number of writings in an atomic operation is the number of edits to the message that keeps the original code point as the closest code. This means that at any time during the write, the reader can always perform a decode to get the message before the writer started modifying. Then, if the writer wishes to rollback the changes it has made, it simply performs a decode and writes the result. If the writer wishes to commit its changes, the writer calculates the checksum and writes the checksum.

For the following examples, model the system as having some memory that is nibble (4 bits) addressable, and writes to a nibble are atomic. Only one thread writes, and multiple threads read. An example of this might be a collection of devices all connected to a single memory chip that has one write port and many read ports. The writer wishes to be able to update a vector of values atomically from the perspective of the readers.

There is also another property of the code in question that is required. The requirement is during the write of the checksum, no code but the previous and the resulting code are ever closest. How the code is decoded matters to satisfy this requirement. An example of a code that does not satisfy this property is the repetition code with the decode step of taking the majority vote for each symbol from each repetition. On the other hand, an example that does work is taking the majority of the whole message with some tiebreaker

strategy. One downside to this method is that the number of writes that a writer can perform is bounded by error correction code distance.

The following is an example of a majority decode repetition code that does not work. Let us say that the message is $0xABC$, and we will use a 3x repetition. So the coded message is $0xABC, 0xABC, 0xABC$. The writer modifies the value to be: $0x12C, 0xABC, 0xABC$. When the writer wants to commit, the writer starts writing the checksum. Next, is $0x12C, 0x1BC, 0xABC$, and here we have the problem. Now, if a reader reads the state at this point, the reader will see the first nibble with a majority for 1, the second nibble with majority for B, and the third majority for C. The reader then sees $0x1BC$, a value that the writer never intended for the reader to ever see.

On the other hand, a method with repetition coding that does work is whole value majority decoding. That is to say, for this example, take all 3 nibbles as a single value, and prefer first value for tie-breaking. The writer modifies the value to be: $0x12C, 0xABC, 0xABC$. Next is $0x12C, 0x1BC, 0xABC$. Here the reader sees no majority so prefers the first value. Next is $0x12c, 0x12C, 0xABC$, and now the reader sees a majority. The writer then finishes updates.

6.3 Repetition Code Concurrent Buffer Algorithm

For simplicity, the algorithm here uses a simple repetition code. The concepts presented also work with other codes such as Reed-Solomon [44]. This algorithm has an advantage of being relatively simple in both implementation and conceptualization. The algorithm needs a logical clock to deal with the

possibility of more than one write in-between a reader's read. Only the writer is going to update the logical clock. So, depending on hardware specifics, the maximum amount of synchronization support needed is a memory barrier to ensure the writers updates become visible to the readers. This is to say, no compare and swap, or other atomic operation support is needed. Since this implementation is relatively simple, it could easily be performed in hardware. For example, consider an FPGA connected to a memory bank on a shared bus. The FPGA wishes to perform a mutation operation involving multiple write operations on a buffer in the memory and the operation should appear atomic to all those reading. This algorithm is simple to implement in this sort of situation. The algorithm keeps two logical clocks, *start_clock* and *end_clock*. The code word is stored in *data* where $data[i]$ is the ith repetition. This algorithm makes the assumption that the logical clocks are updated atomically.

Algorithm 9 Repetition Code Write Algorithm

```
1: function BeginWrite
2:     start_clock ← start_clock + 1;
3: end function
4: function Write(index, value)
5:     data[0][index] ← value;
6: end function
7: function EndWrite
8:     for i = 1 to num_repetitions do
9:         data[i] ← data[0];
10:    end for
11:    end_clock ← begin_clock;
12: end function
```

The write algorithm is shown in Algorithm 9. To start a write session,

Algorithm 10 Repetition Code Read Algorithm

```
 1: function READ
 2:     start ← end_clock;
 3:     result ← data[0];
 4:     if start = start_clock then return result;
 5:     end if
 6:     repeat
 7:         start ← end_clock;
 8:         votes ← data;
 9:     until start + 1 ≥ start_clock
10:     if votes[0] = votes[2] or votes[1] = votes[2]; then
11:         return votes[2];
12:     else
13:         return votes[0];
14:     end if
15: end function
```

the writer calls BEGINWRITE which increments the *start_clock*. Then the writer performs the writes on the first repetition. Finally, when the writer is ready to commit, the writer calls ENDWRITE. The ENDWRITE function first copies the first repetition to the second then copies the first to the third. This order of update must be consistent and known to the readers. Now all repetitions are equal, copy the logical clock value from *begin_clock* to *end_clock*.

The read function for the repetition code concurrent buffer is shown in Algorithm 10. The reader first reads the *end_clock* and then reads the first repetition. After the read of the repetition is complete, read the *start_clock* and compare it to the previously read value of the *end_clock*. If the values are equal, return what was read. Otherwise, reread the *end_clock* then read the whole code word until the read *end_clock* is either equal to the *start_clock* or

one less than the *start_clock*. Then, check to see if the last repetition is equal to one of the other two repetitions. If so, return the last repetition, otherwise, return the first repetition.

Lemma 14. *Given only one writer at a time that follows the update procedure in Algorithm 9, Algorithm 10 will always return either the value before a write has started or after a write has finished.*

Proof. If the read returns on line 4 then there is no ongoing write and the read was a correct value. Once the condition on line 9 is satisfied, this means at most one write is currently ongoing. Because the writer updates the repetitions in order, there are four possible states for the values in *votes*. The first possible state is that all repetitions are the same, an ongoing write caused the algorithm to take the second path but already finished. The second state is the first repetition is modified. In this case, the second and third repetition will be equal so the algorithm returns the third repetition. The third state is the first repetition has the next value that the writer is committing and the second repetition is partially modified. The last repetition is not equal to either the second nor the first; so, return the first repetition as the writer has finished modifying the first repetition. The forth possible case is the third repetition is partially modified. In this case, the first and second repetitions will be equal, so return the first repetition.

□

The following are some reasons why one might choose to use this algorithm. Readers can never delay the writer. At most a memory barrier for synchronization is needed. Relatively simple implementation. Can perform a partial read of the buffer.

One downside to this algorithm is that the worst case scenario for read is that a writer comes along and writes in between every time the reader reads one of the repetitions. This would cause the reader to restart indefinitely, making no progress. Worst case number of read operations is not bounded. It is possible for frequent writes to delay readers indefinitely. Write transactions require three times as many writes. Several of these reasons are complete deal breakers under some circumstances. In most modern computational loads, the main bottleneck is often memory bandwidth. Increasing the number of write and read operations increases the memory bandwidth used.

6.4 ECC Concurrent Buffer Algorithm

The repetition code is rather inefficient in terms of memory usage. One of the major downsides as outlined above is that the repetition code algorithm suffers from an increase in memory bandwidth usage. As such, this section considers the use of a Reed-Solomon error correction code with this approach. In the repetition code, the "checksum" is twice the size of the message. Using an error correction code a "checksum" much smaller can be used. There are two immediately apparent downsides. Performing the encode and decode steps are much more expensive. And, partial reads of the buffer

cannot be done, the whole buffer must be read. The algorithm assumes that a *systematic* code is being used. A *systematic* code has as the first part of the code word, the original message, so such a code can be split into two pieces, the data and the error correction checksum(ecc). There are two logical clocks, *writer_clock* and *commit_clock*. There are routines specific to the code chosen. The first, UPDATE_ECC which updates the error correction checksum for the given modification to the data. A choice of a linear code makes the UPDATE_ECC simply a linear combination of the old value with the new value. And, DECODE which performs the decoding of the error correction code. The DECODE in the pseudo code updates the code word in place and returns true on success. The code word is the concatenation of *data* with *ecc*. Note that one wants to choose a code such that the minimum distance of the code is two times plus one the expected maximum number of edits a writer will perform. If a writer never updates more than the floor of one half the minimum distance minus one, then the reader only ever needs to retry if it happened to read while a writer is currently writing a new *ecc* value.

The write functions are shown in Algorithm 11. To begin a write, the writer makes a private copy of the *ecc* value for it to edit. Then every element the writer updates, the writer increments the *writer_clock* and updates the *writer_ecc*. The writer clock is used to let the reader skip decode if the reader can, and to also know if decode can succeed. It could be that more than one write operation happens between when the reader starts the read and finishes the read of the code word. When the writer wishes to commit what it as

Algorithm 11 Error Correction Code Write Operations

```
 1: function BeginWrite
 2:     write_ecc ← ecc;
 3: end function
 4: function Write(index, value)
 5:     writer_clock ← writer_clock + 1;
 6:     data[index] ← value;
 7:     update_ecc(write_ecc, index, value);
 8: end function
 9: function EndWrite
10:     for i = 0 to ecc_length do
11:         writer_clock ← writer_clock + 1;
12:         ecc[i] ← writer_ecc[i];
13:     end for
14:     commit_clock ← writer_clock;
15: end function
```

written and make it visible to the readers, the writer then copies its updated *writer_ecc* to *ecc* updating the *writer_clock* along the way. And finally, the writer sets the *commit_clock* to the value of *writer_clock*.

The error correction code read function is shown in Algorithm 12. To perform a read, the reader first reads the *commit_clock*. Then the writer reads the code word, in this case is the *data* and *ecc* pair. When reading the code is done the reader reads the *writer_clock*. If the two clock values read are the same, then no concurrent write happened during the read operation, so just return the data. Otherwise, if the number of mutations to the data during the read does not exceed *max_errors* then attempt to decode. For most codes, *max_errors* is the number of errors the error correction code can reliably detect. If the decode operation did not succeed, for example because

Algorithm 12 Error Correction Code Read Operation

```
 1: function READ
 2:     loop
 3:         start_clock ← commit_clock;
 4:         read_data ← data;
 5:         read_ecc ← ecc;
 6:         end_clock ← writer_clock;
 7:         if start_clock = end_clock then return read_data;
 8:         else if end_clock − start_clock < max_errors then
 9:             if DECODE(read_data, read_ecc) then
10:                 return read_data;
11:             end if
12:         end if
13:     end loop
14: end function
```

there was enough modifications that error correction could not be done but enough to detect that there where errors, then go back to the beginning.

6.5 Atomic Swap Performance Comparison

This section presents performance comparisons for a simple application that performed a compare and swap like test. In many applications, a common operation is to have a struct or buffer that is protected by a mutex and accessed by multiple threads. This comparison tries to model the situation where multiple threads want to read a data structure, while one thread performs atomic updates on the data. The readers read the entire contents of the array and validate that it is consistent. The writes "atomically" swap two random elements in the array. Algorithm 13 shows pseudo code for the swap test.

Algorithm 13 Swap Test for ECC Comparison

```
1: function Read Thread
2:     for i = 0 to iterations do
3:         start performance counter;
4:         data ← Read;
5:         end performance counter;
6:         verify(data);
7:     end for
8: end function
9: function Write Thread
10:    for i = 0 to iterations do
11:        x_1 ← rand([0, sizeof(data)]);
12:        x_2 ← rand([0, sizeof(data)]);
13:        start performance counter;
14:        BeginWrite();
15:        a ← data[x_1];
16:        Write(x_1, data[x_2]);
17:        Write(x_2, a);
18:        EndWrite();
19:        end performance counter;
20:    end for
21: end function
```

Three different types of concurrent buffer algorithms where used with the test. The first is lock based, that is to say, have a mutex that is acquired for each access. The second is a copy on write buffer that uses compare and swap atomic routines to swing a pointer for update. The third uses a replication code buffer. In testing, the relative performance of the algorithms did not change much for reasonable values for iterations and number of writers.

Violin plots the distribution of the transaction run times for the three algorithms separated by reader routine and swap routine are in Figure 6.1. The bar in the middle is the average. As can be seen, the lock based approach is quiet a bit slower than the other two on average; but, its fastest was almost the same. It is interesting to note that the distributions appear to be piece wise exponential. While the read range is much the same for all algorithms, the write range is noticeably different. The atomic swap based method is an order of magnitude faster on average than the lock based method. The repetition code method's performance is very similar to the lock free; but, has a smaller range. This indicates that, if it is important that the write algorithm take a fixed constant amount of time, the best choice would be the repetition code algorithm.

Some things to note are the relative number of memory accesses and memory usage between the different algorithms. The base is the lock based algorithm. The lock based has the minimum number of memory accesses and usage. The lock free algorithm is done by copy on write and pointer swing with a compare and swap instruction. So, the number of memory transactions

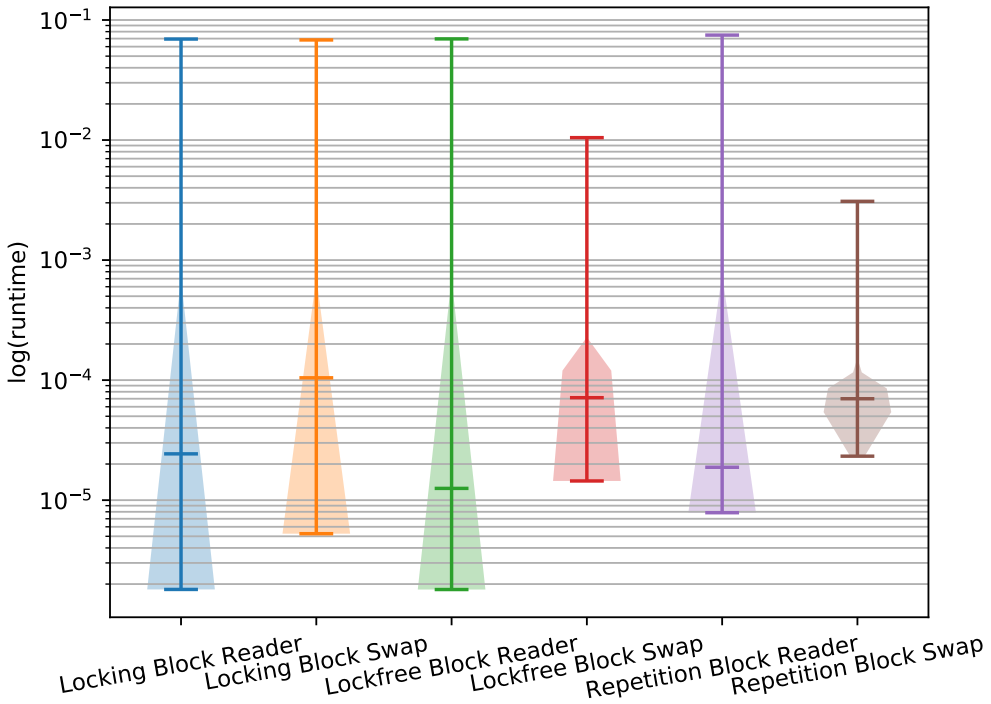


Figure 6.1: Atomic Swap Comparison

is base plus twice the buffer length for the copy and twice the memory usage of the lock based. The repetition code algorithm performs the base plus two writes and two reads of the entire buffer and uses three times as much memory as the base. This could be reduced some, if the writer kept track of the modified values. The repetition code algorithm takes up to three time as many read operations for a read as the other two.

6.6 Hash Map Performance Comparison

A simulated hash map was created that used the three memory block types used above as the hash map buckets. These three block hash maps are compared to the concurrent hash map from the Intel threading building blocks library.

The simulation run was to spawn 20 threads and have each thread randomly and independently perform lookups with 99 percent probability, inserts and deletes with 1 percent probability using a predefined list of keys. The hash map was instantiated with space to fit all the predefined keys; and, then pre-filled to 25 percent capacity before the test was run.

The results of the hash map simulation are in Figure 6.2. As can be seen, the lock based method has the best overall average performance. The lock free version has the worst performance. The worst case run time is about the same for all but the lock free algorithm. I would claim from this that the repetition based algorithm does not perform significantly worse than the fastest algorithm.

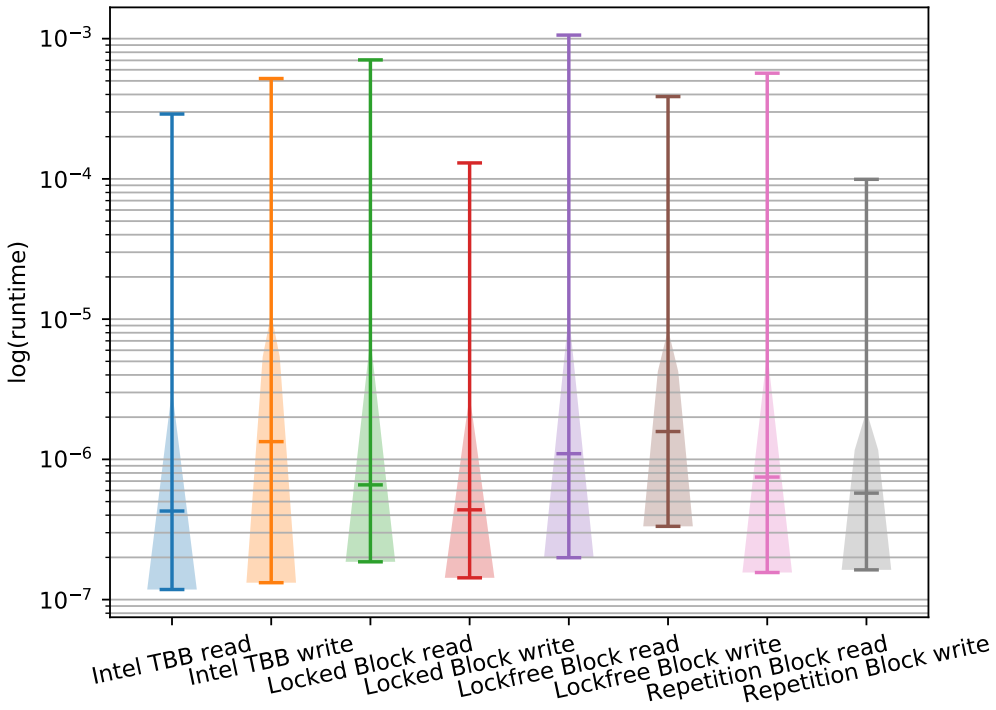


Figure 6.2: Hash Map Comparison

6.7 Conclusion

This chapter has shown how forward error correction codes can be used for synchronizing memory access. The basic building block shown here can be used to build things like B-trees and transactional memory systems. It is a surprising result that the repetition code based algorithm did not perform significantly worse than the alternatives. Common expectation would be to see a serious performance decrease with the extra work. This approach is most useful when the application requires the writer to take a constant number of instructions and have very tight performance envelope. On the other hand, this approach suffers from additional memory access overhead making it unsuitable for some applications.

Bibliography

[1] Ittai Abraham, Danny Dolev, Rica Gonen, and Joe Halpern. Distributed computing meets game theory: robust mechanisms for rational secret sharing and multiparty computation. In *Proc. Annu. ACM symp. on Principles of Distributed Computing*, PODC '06, pages 53–62, New York, NY, USA, 2006. ACM.

[2] Naama Ben-David, Guy E. Blelloch, Yihan Sun, and Yuanhao Wei. Efficient single writer concurrency. *CoRR*, abs/1803.08617, 2018.

[3] Michael Ben-Or, Danny Dolev, and Ezra N. Hoch. Simple gradecast based algorithms, September 2010.

[4] P. Berman, J.A. Garay, and K.J. Perry. Towards optimal distributed consensus. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 410–415, 30 1989.

[5] Piotr Berman and Juan A. Garay. Asymptotically optimal distributed consensus (extended abstract). In *ICALP '89: Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, pages 80–94, London, UK, 1989. Springer-Verlag.

[6] Gabriel Bracha. An $O(\log n)$ expected rounds randomized Byzantine generals protocol. *Journal of the ACM*, 34(4):910–920, October 1987.

[7] John Bridgman and Vijay K. Garg. Brief announcement: all-to-all gradecast using coding with byzantine failures. In Darek Kowalski and Alessandro Panconesi, editors, *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 95–96. ACM, 2012.

[8] John F. Bridgman and Vijay K. Garg. All-to-all gradecast using coding with byzantine failures. Technical Report TR-PDS-2012-001, Parallel and Distributed Systems Laboratory, The University of Texas at Austin, 2012.

[9] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI*, pages 173–186, 1999.

[10] K. L. Clarkson, David Eppstein, Gary L. Miller, Carl Sturtivant, and Shang-Hua Teng. Approximating center points with iterated radon points. In *Proceedings of the ninth annual symposium on Computational geometry*, SCG '93, pages 91–98, New York, NY, USA, 1993. ACM.

[11] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin. Making byzantine fault tolerant systems tolerate byzantine faults. In *Symp. on Networked Systems Design and Implementation*, April 2009.

[12] Brian A. Coan and Jennifer L. Welch. Modular construction of a byzantine agreement protocol with optimal message bit complexity. *Information and Computation*, 97(1):61 – 85, 1992.

[13] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proc. Symp. on Operating Systems Design and Implementations*, pages 177–190, Seattle, Washington, November 2006.

[14] Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and William E. Weihl. Reaching approximate agreement in the presence of faults. *J. ACM*, 33(3):499–516, May 1986.

[15] Danny Dolev, Ruediger Reischuk, and H. Raymond Strong. Early stopping in byzantine agreement. *JACM*, 37(4):720–741, October 1990.

[16] Danny Dolev and H. Raymond Strong. Polynomial algorithms for multiple processor agreement. In *Proceedings of the ACM symposium on Theory of computing*, pages 401–407, New York, NY, USA, 1982. ACM.

[17] Herbert Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, 1987.

[18] A D Fekete. Asymptotically optimal algorithms for approximate agreement. In *Proceedings of the fifth annual ACM symposium on Principles of distributed computing*, PODC '86, pages 73–87, New York, NY, USA, 1986. ACM.

[19] Paul Feldman and Silvio Micali. Optimal algorithms for byzantine agreement. In *Proceedings of the twentieth annual ACM symposium on The-*

ory of computing, STOC '88, pages 148–161, New York, NY, USA, 1988. ACM.

[20] Pesech Feldman and Silvio Micali. An optimal probabilistic protocol for synchronous byzantine agreement. *SIAM Journal on Computing*, 26(4):873–933, 1997.

[21] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[22] Matthias Fitzi and Ueli M. Maurer. Efficient byzantine agreement secure against general adversaries. In *DISC '98: Proceedings of the 12th International Symposium on Distributed Computing*, pages 134–148, London, UK, 1998. Springer-Verlag.

[23] Roy Friedman, Achour Mostéfaoui, Sergio Rajsbaum, and Michel Raynal. Asynchronous agreement and its relation with error-correcting codes. *IEEE Trans. Computers*, 56(7):865–875, 2007.

[24] Juan A. Garay and Yoram Moses. Fully polynomial byzantine agreement in t + 1 rounds. In *Proc. Annu. ACM Symp. on Theory of Computing*, pages 31–41, New York, NY, USA, 1993. ACM.

[25] Vijay K. Garg and John Bridgman. The weighted byzantine agreement problem. In *25th IEEE International Symposium on Parallel and Dis-*

tributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings, pages 524–531. IEEE, 2011.

[26] Vijay K. Garg, John Bridgman, and Bharath Balasubramanian. Accurate byzantine agreement with feedback. In Antonio Fernàndez Anta, Giuseppe Lipari, and Matthieu Roy, editors, *Principles of Distributed Systems*, pages 465–480, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[27] Vijay K. Garg, John Bridgman, and Bharath Balasubramanian. Accurate byzantine agreement with feedback. In Cyril Gavoille and Pierre Fraigniaud, editors, *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC 2011, San Jose, CA, USA, June 6-8, 2011*, pages 215–216. ACM, 2011.

[28] Joseph Y. Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *CoRR*, cs.DC/0006009, 2000.

[29] Jean-Michel Hélary, Michel Hurfin, Achour Mostéfaoui, Michel Raynal, and Frederic Tronel. Computing global functions in asynchronous distributed systems prone to process crashes. In *International Conference on Distributed Computing Systems*, pages 584–591, 2000.

[30] Martin Hirt and Ueli Maurer. Complete characterization of adversaries tolerable in secure multi-party computation (extended abstract). In

PODC '97: Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing, pages 25–34, New York, NY, USA, 1997. ACM.

[31] Satyen Kale. *Efficient algorithms using the multiplicative weights update method*. PhD thesis, Princeton University, Princeton, NJ, USA, 2007. AAI3286120.

[32] Valerie King and Jared Saia. Breaking the $O(n^2)$ bit barrier: scalable byzantine agreement with an adaptive adversary. In *Proc. ACM symp. on Principles of Distributed Computing*, PODC '10, pages 420–429, New York, NY, USA, 2010. ACM.

[33] Thijs Krol. Interactive consistency algorithms based on voting and error-correcting codes. In *TwentyFifth International Symposium on Fault-Tolerant Computing, Digest of Papers, FTCS-25 Silver Jubilee, IEEE Computer Society Press, Los Alamitos*, pages 89–98, 1995.

[34] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. of the ACM*, 21(7):558–565, 1978.

[35] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4:382–401, July 1982.

[36] Kok-Wah Lee and Hong-Tat Ewe. Performance study of byzantine agreement protocol with artificial neural network. *Inf. Sci.*, 177(21):4785–

4798, 2007.

[37] Guanfeng Liang and Nitin H. Vaidya. Error-free multi-valued consensus with byzantine failures. *CoRR*, abs/1101.3520, 2011.

[38] Nick Littlestone and Manfred K. Warmuth. The weighted majority algorithm. *Inf. Comput.*, 108:212–261, February 1994.

[39] Hammurabi Mendes and Maurice Herlihy. Multidimensional approximate agreement in byzantine asynchronous systems. In *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*, STOC '13, pages 391–400, New York, NY, USA, 2013. ACM.

[40] Parchive: Parity archive tool. `http://parchive.sourceforge.net/`.

[41] Arash Partow. Schifra reed-solomon error correcting code library. `http://www.schifra.com/`.

[42] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27:228–234, April 1980.

[43] Michael O. Rabin. Randomized byzantine generals. In *Foundations of Computer Science*, pages 403–409, 1983.

[44] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[45] Ron M. Roth. *Introduction to coding theory.* Cambridge University Press, 2006.

[46] T. K. Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2:80–94, 1987. 10.1007/BF01667080.

[47] Lewis Tseng and Nitin H. Vaidya. Asynchronous convex consensus in the presence of crash faults. *CoRR*, abs/1403.3455, 2014.

[48] Nitin H. Vaidya and Vijay K. Garg. Byzantine vector consensus in complete graphs. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 65–73, New York, NY, USA, 2013. ACM.

[49] S. C. Wang and S. H. Kao. A new approach for byzantine agreement. In *Proceedings of the The International Conference on Information Networking*, page 518, Washington, DC, USA, 2001. IEEE Computer Society.